NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

THE SAAM ARCHITECTURE:
ENABLING INTEGRATED SERVICES

by

Dean J. Vrable
John W. Yarger

September 1999

Thesis Advisor: Geoffrey Xie
Second Reader: Debra Hensgen

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
September 1999

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE THE SAAM ARCHITECTURE: ENABLING INTEGRATED SERVICES

5. FUNDING NUMBERS

5. AUTHOR(S)
Vrable, Dean J., Yarger John W.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*
Computer networks of today are based predominantly on the TCP/IP protocol suite that provides best effort service. The current IP protocol excels in its simplicity and network fault tolerance. The way it implements this simplicity and fault tolerance, however, limits the protocol's ability to provide a guaranteed Quality of Service (QoS). A first step in providing this QoS is to incorporate the concept of flow based routing. This thesis describes an implementation of the Server and Agent based Active network Management (SAAM) system architecture that incorporates flow-based routing. The architecture contains servers that maintain a database that is used for assigning each flow to a path that will provide the needed QoS.

14. SUBJECT TERMS Next Generation Internet, Integrated Services, Quality of Service, Flows, Networks, Routing

15. NUMBER OF PAGES 429

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

THE SAAM ARCHITECTURE: ENABLING INTEGRATED SERVICES

Dean J. Vrable
Captain, United States Marine Corps
B.S. University of Michigan, 1992

John W. Yarger
Captain, United States Marine Corps
B.S., United States Naval Academy, 1993

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1999

Author: Dean J. Vrable

John W. Yarger

Approved by: Geoffrey Xie, Thesis Advisor

Debra Hensgen, Second Reader

Dan Boger, Chairman
Department of Computer Science

ABSTRACT

Computer networks of today are based predominantly on the TCP/IP protocol suite that provides best effort service. The current IP protocol excels in its simplicity and network fault tolerance. The way it implements this simplicity and fault tolerance, however, limits the protocol's ability to provide a guaranteed Quality of Service (QoS). A first step in providing this QoS is to incorporate the concept of flow-based routing. This thesis describes an implementation of the Server and Agent based Active network Management (SAAM) system architecture that incorporates flow-based routing. The architecture contains servers that maintain a database that is used for assigning each flow to a path that will provide the needed QoS.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

The authors would like to acknowledge the financial support of the Defense Advanced Research Projects Agency and the National Aeronautics and Space Agency for the purchase of the equipment used in this thesis.

We want to thank Dr. Geoffrey Xie for his constant willingness to assist us with our research. His guidance and enthusiasm helped us to carry this project to completion. His efforts in arranging for DARPA and NASA funding for this project allowed us to have the resources that were needed to effectively carry out this thesis.

We also want to thank Dr. Debra Hensgen, whose knowledge of distributed systems helped to ensure that this project was a success. Her great demeanor and openess to discussing some of the difficult aspects of this effort are greatly appreciated.

We would also be remise if we failed to acknowledge the immense contribution of Mr. Cary Colwell. His extensive background, depth of knowledge, and willingness to help us contributed greatly to the success of this thesis. He also developed the demonstration station code that was used in the testing and integration.

We also are grateful to our wives, Paulette Vrable and Sarah Yarger, whose love and support were essential during our development of this model and the writing of this thesis. Their help and patience throughout our two years of study are truly appreciated.

We would also like to thank our parents, Jim and Ellie Vrable and Tom and Dianne Yarger, without whose love and sacrifice neither of us would have the opportunities that we are both enjoying today.

I. INTRODUCTION

It appears as though the magic of the Internet has touched just about every segment of our population. Both old and young are now enjoying access to the wealth of knowledge that has been built over the years. As computer networks are increasingly used, one frustrating factor shows through. When people try to communicate using voice and/or video, they often find themselves frustrated by the poor quality of reception. A person's voice is understandable for a while and then stops for a few seconds and then picks up later in the conversation. Conditions like these are caused by a lack of network Quality of Service (QoS).

This thesis takes some well established principles of computer science to suggest an original method to provide the QoS that is needed. Specifically, we propose that a service be implemented within each region of the network that will collect basic topology information and network performance characteristics. With this database of collected information, the service is able to assign a flow that requests a particular QoS to a path. By simply using this type of architecture, numerous enhanced capabilities become possible, such as:

- load balancing
- rapid flow rerouting, as necessary
- display of current network conditions

This chapter begins with a summary of the problem in more detail in order to provide the background that will help one to understand the principles being dealt with. The goal, scope and major contributions of the thesis are described after this background material. Finally, we describe the organization of the entire thesis.

A. BACKGROUND

The desire of people to communicate has been the key to the explosive growth of the Internet. Originally, this information primarily took the form of text and pictures. In many situations, it is more effective to communicate using voice and video. As broadband

communications become more available, the desire to use these is increasing and ability to provide them are becoming more realistic [1,2].

1. Packaging Data for Transport in Flows

The way in which data is packaged depends on what type of network the data is crossing. In the Internet, the data is packaged into a variable length packet format. This packet format consists of a header and payload. A packet header contains delivery information, while the payload contains the actual data.

When passing a large amount of data on the Internet, most applications will utilize TCP, a transport layer protocol. TCP provides reliability by negotiating a session between the source and destination host. A flow is an extension of this idea of a session. In this thesis, a flow is used to describe the sequence of packets from a long duration video or voice session.

2. Requirements for Delivering Voice and Video

The key considerations for transporting voice and video over a network are latency and delay jitter. Latency is the end-to-end delay that data experiences as it moves across the network. Jitter is the variability (in effect, the standard deviation) of the latency in the network.

An uncongested network with wire speed switching and routing nodes will demonstrate no significant latency other than that resulting from the propagation speed of data, which is generally a large fraction of the speed of light. Such a network would display almost no jitter. The obvious downside to allocating each user their own uncongested network is that some fraction of the network's throughput capacity is wasted. Network engineers will often attempt to avoid congestion by provisioning at least twice as much capacity as the expected traffic will demand. This type of over-provisioning "wastes" 50 percent or more of the channel capacity.

3. Managing Congestion with Queues

Switches and routers use buffering to alleviate congestion in a network. Excess traffic is held in queues until the congestion subsides. By creating multiple queues, a switch or router can support different traffic types, each with their own priority. If these queues are not managed carefully, unpredictable latencies can occur. Unpredictable latencies, by definition, cause severe jitter. Additionally packets be dropped when buffer space allocated to a queue is exhausted.

B. GOAL OF THE SAAM PROJECT

The goal of the Server and Agent Active network Management (SAAM) project, under which this research is conducted, is to find a solution that will provide a guaranteed QoS while still maintaining the simplicity and robustness of the underlying TCP/IP architecture. SAAM seeks to provide this QoS by introducing an additional network service that, when requested, can assist applications by reserving the necessary network resources. The SAAM architecture is designed to allow network engineers to incrementally replace existing internal infrastructure. Incremental introduction requires that the SAAM cooperate with this existing internal infrastructure in terms of protocols.

C. SCOPE OF THIS THESIS

The primary goal of this thesis is to refine SAAM's architectural design which previously only existed in a high-level form. This refinement is being accomplished by building and testing a working prototype of that architecture. The implementation of the two major components was divided up between the authors. Dean Vrable implemented the router while John Yarger implemented the server. The entire SAAM team aided the authors in the goal of architectural refinement.

With the addition of a flow identification field, IPv6 is well positioned to support a protocol that provides QoS on a per-flow basis. Countering this advantage, however, is the current difficulty of gaining access to the router source code. To address both of these

issues, an emulated router was constructed. The emulated router routes IPv6 packets on our existing IPv4 network. This emulated router was designed to maintain a per-flow routing table as well as an ARP cache table. Furthermore, the router was implemented to permit the use of resident agents. Resident agents allow components of the router to be upgraded dynamically over the network. Software interfaces that allow for the creation of network applications that reside on top of the router's transport services were also defined. The router manages the communication of these network applications using the standard concept of transport layer (layer 4) port numbers.

The SAAM server was designed to act as one of these network applications. This server implements the two primary tasks of the SAAM architecture. The first task is the maintenance of a relatively accurate status of the current network load. This status is placed in the Path Information Base (PIB) using both hello messages and link state advertisement (LSA) messages. The server's second task is to respond to flow requests. This response is accomplished by searching the PIB in order to identify a good path for the flow. If a supportable path can be found, then the routers that are in the path are sent information so they can update their flow routing tables. Finally the requesting application is notified of its assigned flow id, if the flow can be supported.

Fault tolerance and security are beyond the scope of this thesis and will be addressed elsewhere. Additionally, this thesis concentrated on the design, implementation, and testing of individual SAAM routers and servers. SAAM's hierachical nature will be addressed in future work that will likely leverage other existing work, such as CORBA and/or DCOM.

D. MAJOR CONTRIBUTIONS OF THIS THESIS

The research conducted to produce this thesis has the potential to be integrated into a system that will benefit every user of the Internet. By developing a supportable model for delivering a guaranteed level of QoS over TCP/IP networks, the delivery of voice and video will become reliable. This benefit is particularly important when considering the combat environments of the future. For the foreseeable future, raw

network bandwidth will be less than the demand in many areas, just as today's highways, in certain areas, are often congested. Ensuring that high priority communications can be supported will be essential to success.

E. ORGANIZATION

This thesis is divided into several chapters.

- Chapter II describes other areas of ongoing research that are also designed to provide QoS.
- Chapter III provides an overview of the SAAM architecture. It explains the goals of the SAAM Project as well as how the general SAAM architecture is designed.
- Chapter IV is devoted to the refinement of the SAAM components that are being implemented in this thesis. This refinement was performed using the Unified Modeling Language developed by Grady Booch, James Rumbuagh, and Ivar Jacobson[3].
- Chapter V describes the prototype of the SAAM router and server.
- Chapter VI describes the steps involved in the integration of the router and server.
- Chapter VII summarizes some quantitative measurements made using the integrated router and server.
- Chapter VIII outlines the lessons that we learned from the development of this architecture and outlines the work that remains to be done in the SAAM Project.

II. RELATED WORK

The need for providing a guaranteed QoS has widely acknowledged, as evidenced by the diverse solutions that are being offered to deliver it. In this chapter, we look at these significant efforts and relate them to the SAAM architecture. First, the ATM architecture is described. Next, those technologies that are designed to evolve the current IP network to provide QoS is examined. These technologies include the Resource Reservation Protocol, Differentiated Services, and TCP Rate Control.

A. ASYNCHRONOUS TRANSFER MODE (ATM)

The basic idea behind ATM is to transmit all information in small, fixed-size packets called cells. ATM networks are entirely connection-oriented. No data is delivered in an ATM network without first establishing a Switched Virtual Circuit (SVC). An ATM network will not establish a virtual circuit for a requested flow unless it can ensure a guaranteed QoS from end to end. When a new flow is not admitted, customers receive a type of "busy signal" indicating that ATM network is too congested. However, if a virtual circuit is assigned to a flow, the same latency and jitter bounds which have traditionally been provided to voice and video traffic in legacy Time Division Multiplexing (TDM) circuits can also be provided in ATM.

ATM promises to provide many of the properties that are needed to guarantee QoS. It also carries with it the baggage of requiring the replacement of existing equipment along the path upon which the QoS is to be guaranteed. While LAN Emulation allows for the connecting of an existing IP network to an ATM network, ATM provides no solution for providing QoS over the IP network part of the path.

In spite of this shortcoming, the ATM forum has designed and analyzed some useful ways of providing guaranteed QoS. One such analysis has produced the Private Network Node Interface (PNNI) protocol. PNNI is a switch-to-switch protocol that is designed to support efficient, dynamic, and scalable routing of SVC requests in a multivendor private ATM environment. PNNI phase I consists of two protocols: routing

and signaling. PNNI routing uses a hierarchial topology protocol to disseminate topology and resource information among participating switches or groups of switches. PNNI signaling uses the topology and resource information available at each switch to construct a designated transit list which determines a path that to meet the requested QoS objectives. PNNI signaling is also used to complete the connection.[4]

1. System Initialization

Each ATM switch exchanges updates with its neighboring switches concerning the status of the links, the resources and the status of these resources, and the identity of each other's neighboring switches. This information is used to build a topological database of the entire network. Each ATM switch in the group will have an identical copy of the topological database. If a change in topology occurs (e.g., link is broken), then only that change is propagated to all of the switches.[4]

The SAAM Project proposes to utilize a similar initialization method. In our model, currently available best effort routing protocols (e.g., IP) will allow the sending of a hello message to a SAAM server. This server may or may not reside on a SAAM router. This server then builds a Path Information Base from these hello messages. If a change in topology occurs, only one message is sent to the server.

2. Scalability

PNNI is designed to be scalable. It achieves this scalability by using a hierarchical routing structure. Information about the topology of a group of nodes is aggregated and presented as a single node in the next level up in the hierarchy. A single node is elected to represent a particular group to that next higher level. By only presenting a summary, this node limits the amount of information about a group of nodes that is advertised. This is a key factor, since ATM networks deals with a huge volume of information about connections. This reduction of complexity is called *topology*

aggregation. Topology aggregation necessitates that some degree of accuracy be sacrificed.[4]

The SAAM architecture will use a similar mechanism to achieve scalability. At the lowest level, a SAAM server will be placed within an autonomous region. The servers at this level will provide services for each router in the region. Each of these servers identifies their adjacent regions and determines the currently available QoS between the two regions. These servers then function as routers themselves to the next higher level server by providing a summary of its physical connections to the adjacent regions as well as the QoS parameters to describe the level of congestion at different service levels.

3. Admission Control

PNNI utilizes source routing to actually reserve a path through a network. The first switch in the SVC request path computes the entire path according to its topological database. During this computation it performs a Generic Connection Admission Control (GCAC) procedure, which provides the originating node with an estimate of whether each switch's local Connection Admission Control (CAC) process will accept the connection. The QoS metric values stored in the topological database allow the first switch to make a good decision about which path to choose. The other switches along the chosen path then simply perform the same CAC function. If the current switch grants admission, it will forward the SVC request to the next switch in the source-route path.[4]

The SAAM architecture calls for the SAAM server to be able to dictate to its member routers to accept flow assignments. This simplifies the system and isolates the admission control operations for the region. In a hierarchy of SAAM servers, this would allow lower level servers to decide the best way to get the flow through its own region.

B. RESOURCE RESERVATION PROTOCOL (RSVP)

In response to the need for a guaranteed bound on latency, the Internet Engineering Task Force (IETF) began an effort to create layer-3 and layer-4 mechanisms

that can be used to ensure QoS for real-time data streams. This Integrated Services effort focused on the development of RSVP. Applications can use RSVP to obtain QoS guarantees from routers, thereby reserving a path with the desired characteristics across an IP network. RSVP accomplishes this by using an admission policy similar to ATM. A connection will be refused by RSVP if it is determined that the resources of the network are inadequate to provide the requested service levels. There are two major differences, however, between RSVP and ATM. First, RSVP records its path across a network to its destination. After reaching its destination, RSVP works backward to establish the required reservations. The second major difference is that RSVP can be implemented in an evolutionary manner, while ATM must be implemented in a revolutionary manner. ATM requires the replacement of existing data network devices to deliver QoS, while RSVP is implemented as a software option within router software.

Although RSVP has been successfully implemented within a single region, it has had problems scaling to the larger Internet backbone. Scalability of the current Internet results from the statelessness of IP routers. RSVP, however, requires routers to maintain state. While this overhead can be easily accommodated on a private network, it becomes a practical impossibility on the Internet core, where traffic volumes are enormous. Therefore it is not surprising that no ISPs have stepped forward to support RSVP on their public networks.

SAAM is similar to RSVP in that it is an evolutionary technology. SAAM is designed as an added service for IP networks that can be implemented gradually through software upgrades. What makes SAAM different from RSVP is that SAAM does not rely on the limited knowledge available at each router to make a local decision as to the best direction on which to forward the RSVP initialization packet. Rather, the SAAM architecture relies on more global knowledge available from a hierarchically structured group of servers that maintain a picture of the current status of the network and make wise assignments for each flow request.

C. DIFFERENTIATED SERVICES (DIFFSERV)

Some members of the IETF recognized that RSVP did not meet the needs of network providers. As a result, a subgroup from the Integrated Services Architecture group was created to propose Differentiated Services (DiffServ) standards. While DiffServ does not guarantee resource availability on a per flow basis; it does establish mechanisms to support QoS for aggregated data.

Routers can only affect service quality at layer 3 through variable queuing. No packet, regardless of priority, however, can traverse a router any faster than at wire speed. All that priority queuing can provide, in the presence of congestion, is the assurance that high priority packets will pass through a router faster than a low-priority packet. Priority queuing alone cannot guarantee low latency or absence of jitter to individual flows.

D. TCP RATE CONTROL

TCP rate control is another way to restrict traffic on IP networks. TCP rate control adjusts end-point TCP window sizes explicitly, rather than permitting them to grow as large as possible as quickly as possible.

Together priority queuing and TCP rate control are able to ensure that a stream of packets does not monpolize the network, but neither can ensure congestion-free traffic flow. These are valuable techniques for operating within a congested network and for prioritizing classes of traffic. By using these techniques, bandwidth availability, latency, and jitter for a particular flow may be restricted to specific limits. These techniques can only restrict congestion under certain congestion levels and traffic mixes. Furthermore, these techniques do not guarantee specific service levels.

III. OVERVIEW OF THE SAAM ARCHITECTURE

The SAAM architecture is designed to support applications requiring a long duration, constant bit-rate flow between two hosts on a TCP/IP network. Before describing the SAAM architecture in detail, however, we present a list of issues that have a direct impact on the feasibility of a server based network architecture. Many of our design choices are based on the understanding of these issues. After these design issues, the two main SAAM components, the router and server, are described. We then shift our focus over to the server's hierarchical structure. Finally, we look at the server's database of path information.

A. DESIGN ISSUES

In this section, we describe how the design issues of responsiveness, scalability, and fault-tolerance have impacted the overall SAAM architecture.

1. Responsiveness

To support integrated services, the network must be able to detect and react to changing conditions, especially QoS degradation along a path, within a short time frame. Therefore, SAAM uses a proactive approach in data collection. Moreover, similar to PNNI's topology aggregation, SAAM should aggregate the data about individual links into "ready to use" path performance information.

2. Scalability

SAAM must be able to scale to provide a complete solution for global networks that consist of hundreds of routers. On one hand, it is desirable to have a small number of servers. On the other hand, there is an upper limit on the number of routers that a server can support. The scalability issue is also very important when determining how frequently a server should update its PIB. More frequent updates will result in more accurate

information. However, they also cause more (computation and communication) overhead on the network and servers.

3. Fault-Tolerance

If not carefully designed, the failure of one SAAM server could have a devastating effect on the performance of the entire network. Therefore, servers must be deployed in such a way that the failure of one server can only affect the performance of a small set of routers for a short period of time. In addition, it should be possible to deploy redundant servers.

B. SAAM ROUTERS AND SERVERS

SAAM consists of light-weight routers and a small set of heavy-weight servers. Logically, each router is a client of a single SAAM server process (see Figure 1).

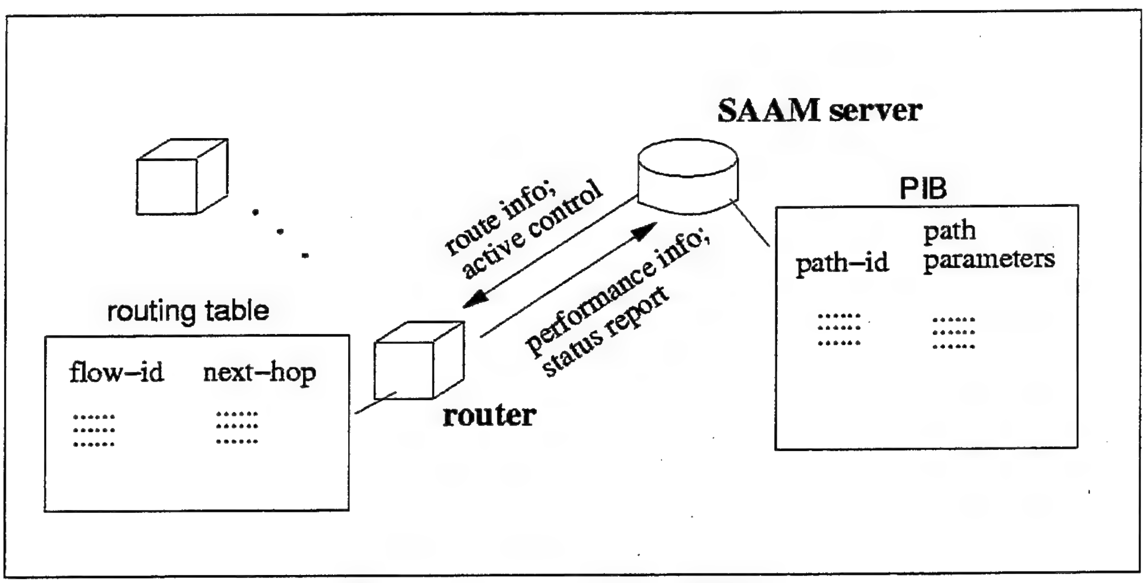


Figure 1. Logical Model of SAAM

Next, we describe how a particular router and the SAAM server process interact in this model. For brevity, we will focus on those aspects related to QoS routing.

SAAM requires (preferably dedicated and real-time) duplex communication channels between each router and its server. We assume that these channels are established when the router joins the network. The router does not participate in QoS routing; it updates its flow-based routing table with route data that is passed down from the server. Note that the router will, however, still participate in conventional routing of non-real-time traffic for backward compatibility.

The SAAM server builds a PIB to support QoS routing. Specifically, the server identifies those paths or subpaths that can potentially be used to route flows, and maintains up-to-date performance parameters for each of them. The server computes path performance parameters by aggregating link level performance data passed up from each router.

C. HIERARCHIAL ORGANIZATION OF SERVERS

If the Next Generation Internet is going to provide any guarantee of QoS, a solution must be found to manage the large number of flows that are expected. More specifically, the SAAM architecture must address how this scalability issue is going to be handled, particularly since it needs to maintain per-flow state[1].

To address the scalability issue, SAAM organizes its servers in a hierarchy. (See Figure 2.) Specifically, at the first level, SAAM partitions the network into regions, and sets up one server for each region. The current approach to network partitioning using Autonomous Systems [5] can easily be extended to perform this task. Once established, the SAAM server can provide path analysis and flow-assignment services on behalf of the routers in its region.

Each server communicates only with its immediate children, which may be routers or servers. A server summarizes the QoS of its region for the next higher level. This summary will allow the server to present itself as a router to the next higher level of servers. These parent servers will simply see that it has a network of routers that may have higher than normal delay and loss rate. Just as an actual router is responsible for

forwarding a flow through its internal switching fabric, a lower level server is responsible for determining how to forward a flow through its internal network.

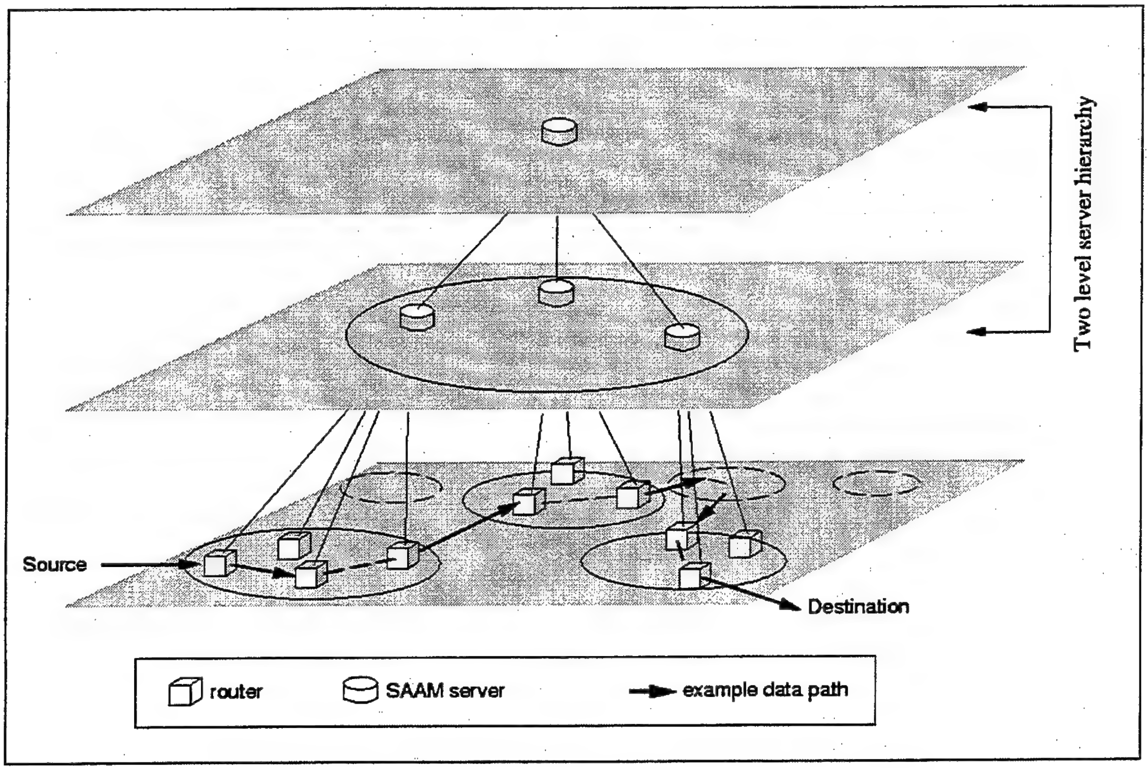


Figure 2. Hierarchical Organization of SAAM Servers

Each server only needs to deal with flows in or across its own region. Additionally, the SAAM architecture is designed to only facilitate resource reservations by real-time, long-duration flows (carrying video, audio, or large data sets). SAAM will not change how current routers handle best-effort traffic. Thus, the number of flow requests that a SAAM server needs to process will only include flows that pass through its region.

Similar to today's architecture, each SAAM region has a subset of routers, called border gateways, through which data can enter and leave the region. SAAM uses a parent server at the top level to enable the establishment of flows between two regions.

Unlike the telephone network, the Internet consists of many independently operated Internet Service Providers (ISPs). One must recognize that there must be

[1] In this thesis, we focus on QoS routing on a per flow basis. SAAM will be extended to support per class

concrete incentives for these ISPs to introduce an architecture like SAAM to their networks. Perhaps the most important incentive for adding the SAAM system to an existing network is the ability to provide more predictable performance for their flows, which should attract more customers. Another major incentive is that by using SAAM's efficient resource allocation methods, an ISP may accommodate more traffic and therefore increase revenues. By coexisting with its existing routing protocols, an ISP may use SAAM without jeopardizing its operations.

This incremental deployment of SAAM servers can also integrate nicely with the Differentiate Services model. The SAAM architecture can help ISPs that support different Per Hop Behaviors (PHBs) to achieve end-to-end QoS. This type of coordination is needed to handle dynamic network load and link failures.

The main advantage of the above architecture is that it allows SAAM to build a scalable PIB. The hierarchical architecture also permits SAAM to be gradually deployed into today's networks. Specifically, SAAM can be implemented initially in one part of a network. The top-level SAAM server will function as a speaker for all routers in the SAAM part of the network, i.e., it will become the sole participant in the information exchange with routers in the other (non-SAAM) part of the network.

D. THE SERVER'S PATH INFORMATION BASE

So far, we have looked at how the server and router interact with each other and how servers form a hierarchical structure to achieve scalability. Next, we will look at how information is managed within the server. As discussed earlier, an essential component of the SAAM architecture is the PIB. When designing a PIB, performance and cost are key issues to consider.

The SAAM server will be able to perform a wide range of network functions such as routing, resource reservation and network management. For the server to perform these functions well, the PIB needs to maintain sufficient information, and to keep this information up to date.

---

routing (required by DiffServ) and even semi-permanent routing.

The cost, or overhead, of building and maintaining the PIB should be carefully analyzed and controlled. In particular, the PIB must scale well as the number of routers in the region grows.

For the remainder of this section, we describe how the server's PIB is designed to take advantage of the SAAM architecture to achieve high performance and to control cost. To illustrate the benefits of this PIB design, we also explain how SAAM can make use of the PIB to perform efficient QoS routing and re-routing.

First, we describe the system model for our PIB design. Specifically, we define a path in the context of an integrated services network, and identify a set of important path parameters that will be managed by the PIB.

1. Service Level Pipe Parameters

In an integrated service network, each network link is shared by a set of logical service level pipes [6, 7, 8], each of which provides a particular level of network performance measured by packet delay and packet loss rate. (See Figure 3.)

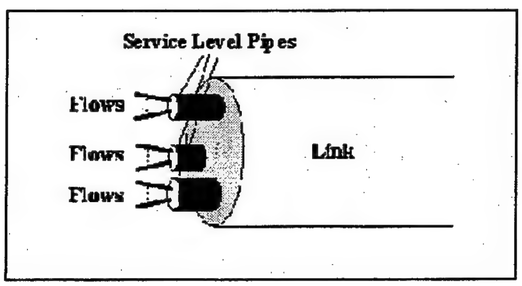


Figure 3. Links Shared by Service Level Pipes

A path is an ordered sequence of service level pipes. Specifically, an arbitrary path (denoted by π) is represented by $\pi = <s_1, s_2, ..., s_K>$ where s_k is the kth service pipe in the path, $k = 1, 2, ..., K$. Each service pipe is characterized by the parameters listed in Table 1.

`s.D`	target upper bound on the total packet delay (in seconds); includes queuing delay, transmission delay and propagation delay
`s.E`	target upper bound on the percentage of packets that incur a delay greater than `s.D`
`s.B`	amount of pre-allocated link bandwidth; in bits per second
`s.R`	bandwidth available for new flows; initially set to `B`

Table 1. Parameters of a Service Level Pipe

2. Path Parameters

The set of parameters that characterize the QoS that a path can provide is listed in Table 2. Most of these parameters are generalizations of what have been defined for a service pipe.

Note that when a rate based packet service discipline (e.g. Weighted Fair Queuing) is used at each service pipe, the target end-to-end delay upper bound of a path can be much smaller than the sum of the target per-hop delay bounds. In such a case, we redefine `s.D` to be a target delay upper bound based on the expected packet arrival time; hence the definition will continue to hold [9,10].

π`.D`	target upper bound on the total packet delay, expressed by π`.D = ` $\sum$ `s.D.`
π`.E`	target upper bound on the percentage of packets that incur a delay greater than π`.D` and thus considered to be lost.
π`.B`	total effective bandwidth, which is expressed by π`.B = min{s.B}.`
π`.R`	currently available effective bandwidth, expressed by π`.R = min{s.R}.`
π`.F`	set of flows that are currently using π.

Table 2. Parameters of a Path

3. Flow Parameters

A flow is a network connection established by an application requiring a certain type and level of network QoS. The flow traverses a path composed of a sequence of service level pipes whose composite QoS meets the specified requirement. The two QoS parameters associated with a flow, `f`, are listed in Table 3. The objective of QoS routing and re-routing is to allocate, and re-allocate if necessary, a statistical path to connect the source and the destination of the flow while satisfying: $\pi.D \leq f.D$ and $\pi.E \leq f.E$.

f.D	the delay bound requirement of a flow
f.E	loss bound requirement of a flow

Table 3. Parameters of a Flow

4. Building the Path Information Base

The cost of building and managing the PIB is controlled by leveraging the hierarchical architecture of SAAM. It suffices for the SAAM server of each region to build and manage a relatively small PIB that contains information for only local paths in the region. Specifically, information for a long-distance path (i.e., one that crosses multiple regions) is built and managed jointly by three SAAM servers. A first-level server is responsible for the source segment (from the source to an outgoing border gateway). Another first-level server is responsible for the destination segment (from an incoming border gateway to the destination). The parent server is responsible for the middle segment between the border gateways. In the remainder of this section, we will focus on how to build a regional PIB.

We also identify and exclude undesirable paths from each regional PIB to reduce the size of the PIB. Specifically, those paths that contain a loop or have a regional hop count greater than a predetermined value H_{max} are deemed invalid. We define a path as valid if and only if it has a hop count less than or equal to H_{max}, and all the routers in the path are distinct.

Next, we describe the content of a PIB and the steps that a SAAM server takes to build its regional PIB. Consider a particular SAAM region and its SAAM server. Assume that there are M routers in the region.

The server takes two major steps to build the PIB. First at network boot-up time, the server assigns a unique index i between 1 and M to each router; and for each router i, it computes and stores the set of directly connected routers:

```
Parents(i) = {j |
     there is a service pipe from router j to router i}
```

Afterwards, the server uses a recursive algorithm[11,12] to build its list of possible paths in its regional PIB. The algorithm searches for new source routers from a final destination to find new paths of increasing length. The Parents(i) array provides the listing of routers that are directly connected to the router, i. The algorithm has an average complexity of $O(M * g^{Hmax})$, where g is the average size of the Parents set for a router. Typically, g does not exceed 3.

IV. MODELING

Modeling is a central part of all of the activities that lead up to the development of good software. As such, the SAAM Project first developed models to communicate the desired structure and behavior of our system. These models helped us to more fully understand the components of the system's architecture and their interaction. Modeling did indeed expose opportunities for simplification and reuse.

This chapter starts off with a requirements analysis of the SAAM Project. We then describe the communications that occur between the router and server. We finish up the chapter by looking individually at the details of the server and router models.

A. REQUIREMENTS ANALYSIS

The SAAM Project has two main components: the router and the server. The router receives packets and then forwards them along the appropriate path based upon the content of the packet's IP header and the router's routing table. The server builds a view of the network from messages sent from the routers and then performs the function of admission control for applications desiring a guaranteed QoS.

We chose to develop our model within the context of a network built on IPv6, which provides a flow identification field. While this field is provided by the IPv6 standard, other than the DiffServ standard, there is no standard way of using it to obtain QoS for flows. This is where SAAM, or a system similar to SAAM, will be needed.

In the subsections that follow, we examine the key requirements of the SAAM architecture. Specifically, we define who the customer is for this development effort and the what the goals of the SAAM architecture are. Finally, we describe the system functions and attributes.

1. Customers

The customers for this product are the Defense Advanced Research and Project Agency and the National Aeronatics and Space Agency who are both funding this basic research to investigate the ideas needed to support integrated services, and to develop proofs of concept and demonstrations. This research will be utilized later by integrators who will eventually develop the ideas derived from this research into actual products.

2. Goals

Our goal in developing this software is to accurately model the proposed SAAM architecture. SAAM's goals include:

- Minimize the overhead of maintaining flow status at each router
- Minimize the overhead of collecting network status
- Determe good paths quickly for assignment to flows
- Reconfigure flows quickly when a path is disrupted

3. System Functions

System functions are what a system is supposed to do. See Table 4 for a list of the basic functions that describe what the SAAM architecture should do. Evident functions are those functions that the user should be congnizant of when they are performed. Hidden functions are those functions that are not visable to users. Primary types of functions are those that are critical to the system's primary function and will be addressed in this thesis. Secondary functions would provide enhanced capabilities to the system.[13]

Ref #	Function	Category	Type
1	Pass topology information to servers	Hidden	Primary
2	Maintain collected topology information	Hidden	Primary
3	Analyze traffic patterns	Hidden	Secondary
4	Respond to queries for a path meeting specified QoS	Hidden	Primary
5	Disseminate resource reservations/routing tables	Hidden	Primary
6	Authenticate administrator/providers of information	Hidden	Secondary
7	Display configuration forms	Evident	Secondary
8	Display network status	Evident	Secondary

Table 4. System Functions

4. System Attributes

System attributes are nonfunctional system qualities. Attributes are distinct from with system functions. They are characteristics or dimensions of the system. System attributes are described with either details or boundary constraints. Attribute details tend to be discrete, fuzzy, symbolic values of the attribute, while attribute boundary constraints are mandatory boundary conditions which is usually expressed on a numeric range of values of an attribute. See Table 5 for a list of the basic system attributes that characterize the SAAM architecture.[13]

Attribute	Details and Constraints
Design	(detail) modular, object oriented
Response Time	(boundary constraint) when notified of significant changes, the flows assigned should be reassigned within 3 seconds.
Interface Metaphor	(detail) forms-metaphor windows and dialog boxes
Fault Tolerance	(boundary constraint) must continue to provide service within 30 seconds in case of power or device failure
Operating System Platforms	(detail) any platform supporting a Java Virtual Machine or a native Java compiler
Ease of Use	(detail) an administrator can configure the system for the first time in under 30 minutes

Table 5. System Attributes

B. COMMUNICATIONS BETWEEN SAAM COMPONENTS

The SAAM router and server must exchange information in order to provide the services of this architecture. Effort was made to minimize the amount of information exchanged in order to cause as little overhead within the network as possible. The information to be exchanged between these components can be viewed either as individual message objects or as physical packets (see Figure 4).

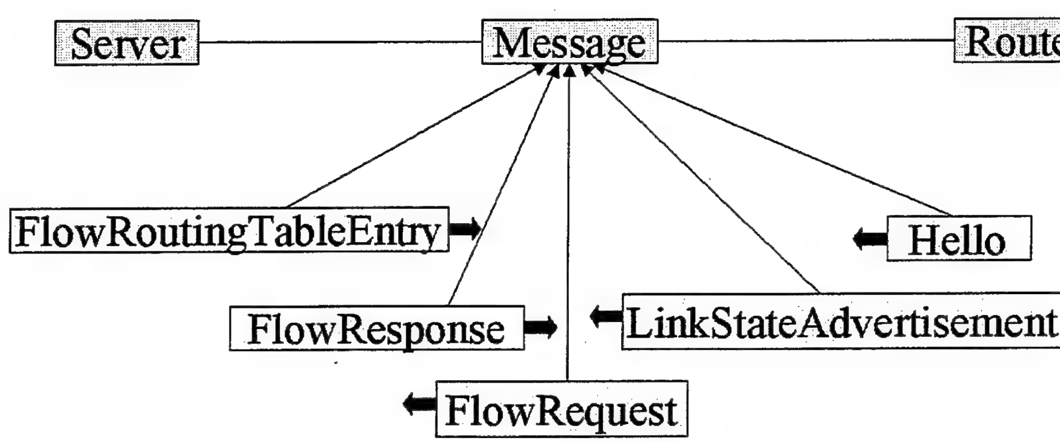


Figure 4. Messages Communicated Between Routers and Servers

1. Hello Message

For the server to be able to construct a view of the network, it must receive certain information from each router that is connected to the network. This topology information is placed within the *hello* message. This message identifies the IPv6 address and bandwidth of each of its interfaces. The server uses these collected hello messages to determine all of the possible paths through the network from each source to each destination.

2. Link State Advertisement Message

Knowing the paths described above would be of little use, if this knowledge did not also include QoS information. In SAAM's initial implementation, we concentrate on two key parameters: the average delay and loss rate experienced by packets in the different outgoing queues, although our design is flexible so as to be able to easily incorporate many additional QoS attributes. The QoS parameters are calculated for each service level pipe of each interface of each router. Each router then communicates this information to the server using *link state advertisement* (LSA) messages. After receiving these LSA messages, the server develops a more accurate view of the effective QoS of those paths that cross the service level pipes described by the LSA.

3. Flow Request Message

While the server is enhancing its understanding of the effective QoS of these paths, it is also capable of responding to requests for the establishment of flows. These requests are sent by applications in the form of *flow request* messages. A flow request message contains a requested delay and loss rate, as well as the amount of bandwidth the application expects to consume.

4. Flow Routing Table Entry Message

If the server determines that the request can be supported, it assigns a flow identifier to the new flow and allocates a good path to it. Before responding to the application, however, the server identifies each of the intervening routers in the path. To each of these routers, it sends a *flow routing table entry* (FRTE) message that indicates what each router should do if it receives a packet with this new flow id. Specifically, a FRTE message contains a flow id and the appropriate next hop and the service level upon which to send the message on.

5. Flow Response Message

After these messages are sent, the server sends a *flow response* message to the requesting application. The flow response simply indicates to the application what flow id should be assigned. If there is no path that can support the flow request, the server simply constructs a flow response that assigns the flow an id that indicates that this QoS cannot currently be supported. The flow would be assigned a flow id of zero, which, by the IPv6 protocol standard, indicates non-flow based (i.e. best-effort) traffic.

C. SERVER MODEL

The server is an application that could reside on a router or on a dedicated host. Its primary function is to develop an accurate picture of the network and then to respond to flow requests. By processing hello messages and LSA messages, it populates a Path Information Base (PIB) (see Figure 5).

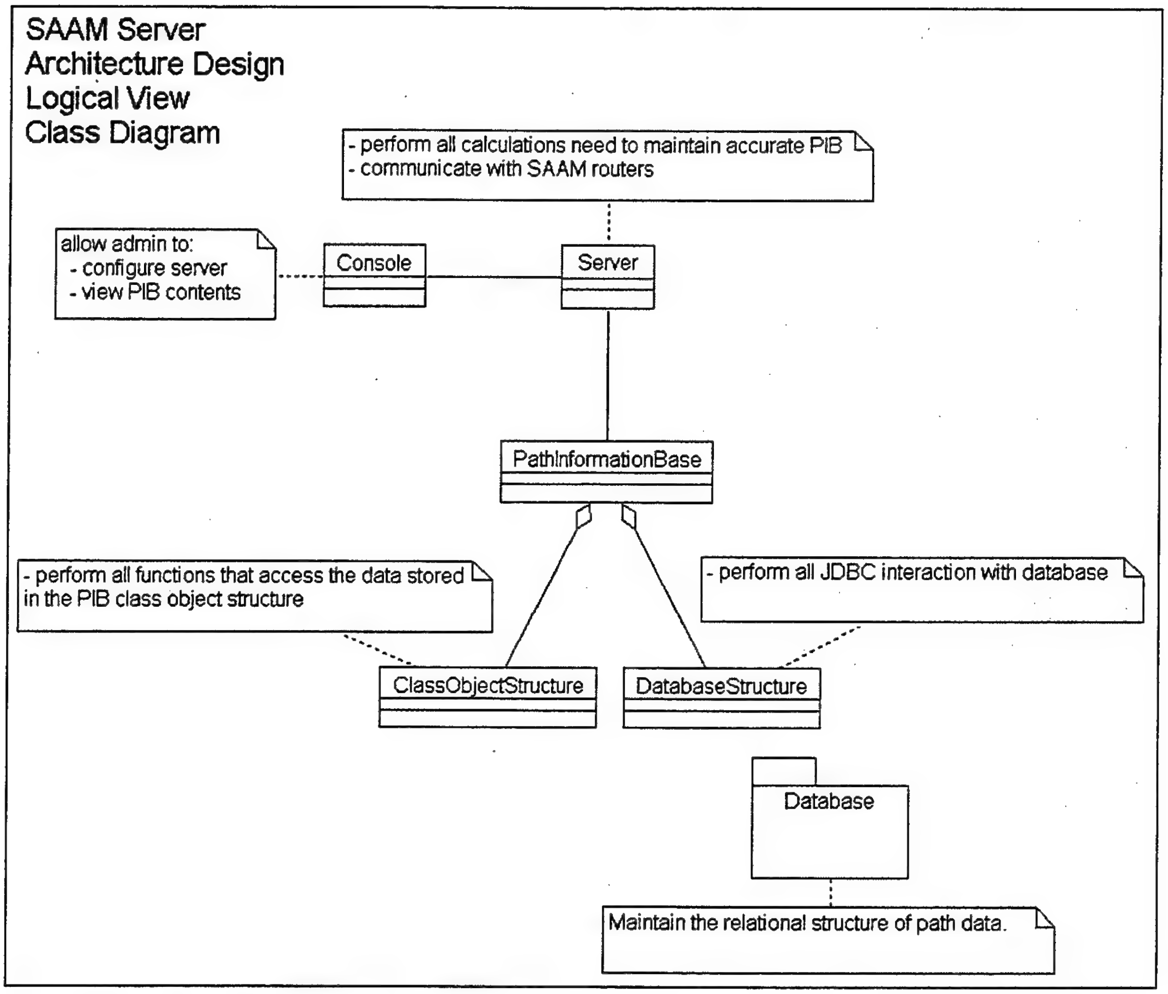


Figure 5. SAAM Server Architecture Design

The core of the PIB is a representation of the physical network. From this representation, the server is able to calculate every physical path through the network. These paths are initialized to have the full bandwidth of the service level pipe with the smallest bandwidth in the path. Initially, these paths are also assumed to have no effective delay or loss. On top of this utopian view of the network, the server will begin to build a more accurate picture of the network by processing LSA messages. As the server gets more up-to-date information from these LSA messages, it recalculates the effective QoS of those paths that cross over these service level pipes.

1. Entity-Relationship Model of the Path Information Base

The PIB is a body of knowledge describing the state of a network. It provides methods that present specific information about the network. It also provides methods for updating and maintaining the status of the network. The server is responsible to ensure that it maintains the integrity of the PIB by passing correct information to it. The data within the PIB can be divided into two basic types of entities: physical and logical (see Figure 6).

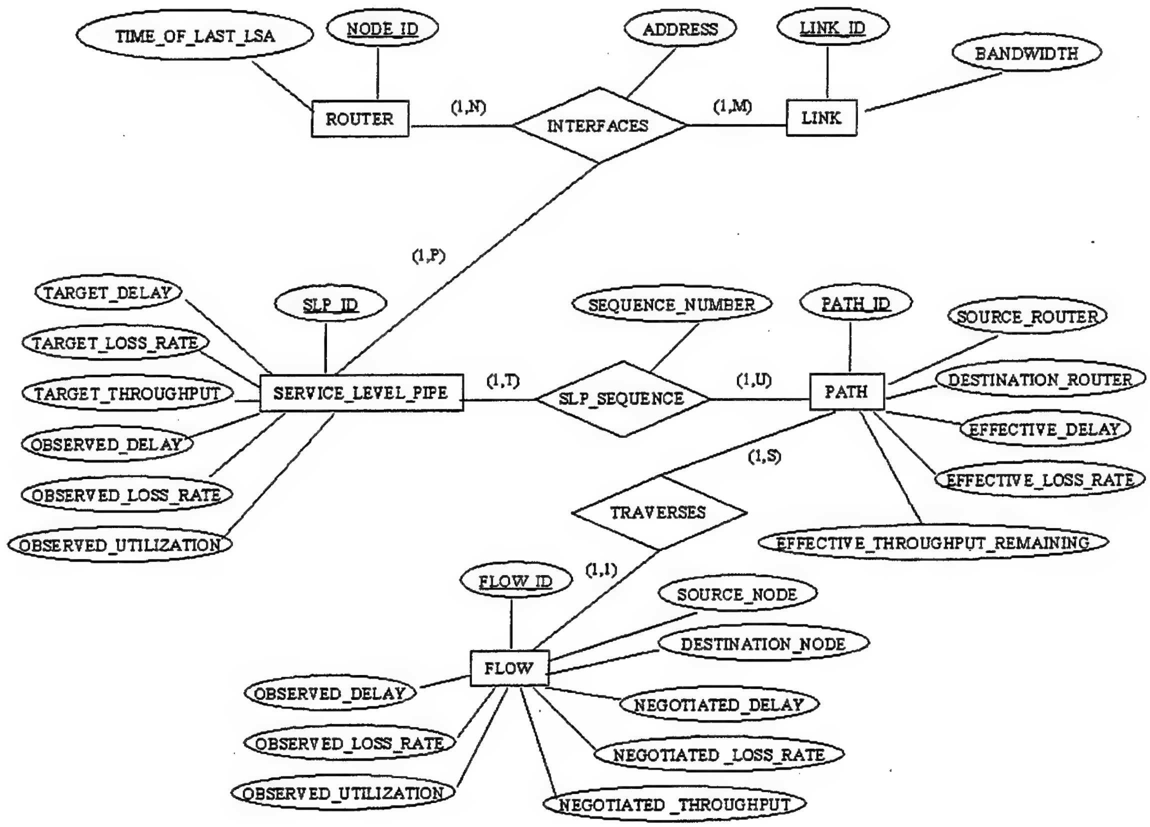


Figure 6. Entity-Relationship Model of the PIB

a) Physical Entities

The physical characteristics of the network that must be collected are grouped according to the two entities that they describe. These entities are the routers and the links. The relationship between these two entities is referred to as an interface. The

attributes of these physical entities can be communicated to a server during its initialization of the router or server in the form of hello messages.

(1) Router. A router connects two or more links and performs layer 3 functions. A router is characterized by a node identification number that is assigned by the server. The time that the last LSA message was sent from this router is also stored in the PIB. This time stamp is needed to determine whether a node is still able to communicate its current status to the server.

(2) Link. A link is a physical media that connects two or more nodes together. A link is characterized by a network address. This network address is derived from the assigned IP addresses of its router interfaces. A link is also characterized by the maximum bandwidth of its physical media and lower level protocol.

(3) Interface. An interface is the connection of a router to a link. The combination of an IP address, the node id of the router, and the network address of the link characterize an interface.

b) Logical Entities

To allow for the assignment of flows, the SAAM server must take the physical representation to determine the possible paths that could be assigned to a flow. Since routers manage congestion and enable prioritization through the use of outgoing queues, SAAM must also take into account the delay and packet loss experienced at the queues that make up the path, if that particular attribute is part of a QoS request. Using an entity-relationship model, we have a service level pipe entity that describes the status of an outgoing queue and a path entity that describes the effective QoS from a source to a destination. The relationship between these two entities consists of the sequence of service level pipes that make up the path.

(1) Service Level Pipe. A physical link is divided into multiple logical service levels. When a service level is combined with an interface, it forms a service level pipe. A service level id and the IP address of an interface can be used to characterize a service level pipe. QoS parameters are monitored through the use of these

service level pipes. It is important to emphasize that routers manage congestion through the use of outgoing queues. Delay and packet-loss occur predominantly at these outgoing queues. This is also where the resultant bandwidth utilization is monitored. In a SAAM router, a unique service level pipe is associated with each outgoing queue. The target QoS and the observed QoS must both be maintained within the PIB for each service level pipe.

(2) Path. A path can be defined as a sequence of service level pipes from a source router to a destination router. By defining a path in this way, SAAM can calculate the effective QoS for the path by summing up the delays and loss rates of the service level pipes that make up the path. SAAM can also determine the effective throughput remaining for each path. The server assigns each path a unique path identification number.

(3) Service Level Pipe Sequence. The relationship between the service level pipe entity and the path entity is the sequence number of a particular service level pipe in a particular path. The primary keys of the service level pipe and the path can be combined to serve as the primary key of the SLP sequence relationship.

These three logical elements (service level pipes, service level pipe sequence number, and paths) are derived from the characteristics of the three physical elements (routers, interfaces, and links). By maintaining information about these logical elements, the server can selectively assign delay sensitive flows to paths using high priority service levels. The queues represented by these service level pipes would be scheduled to ensure that they guarantee the appropriate bound on delay and loss.

(4) Flow. The final entity that must be maintained within the PIB is the flow. A unique flow id, a source node's IP address, and a destination node's IP address characterize the flow entity. The PIB also stores, for each flow, the negotiated delay, loss rate and throughput as well as the observed delay, loss rate and throughput. Finally, the server assigns the flow entity a path id that can provide the requested level of service.

2. Class Object Model of the Path Information Base

The development of a class object model of the PIB involved sustantial tradeoff considerations. Increasing the data retrieval rate increased the amount of memory required to store the data. The design that follows attempts to minimize the amount of memory required to store the data. This decision, however, results in a need for more searches to retrieve some needed data.

a) Nodes

The nodes object is a hash table that uses the assigned node id of the router as the key (see Figure 7). The elements stored in this hash table are references to other hashtables that maintain interface information. The interface hash table uses the IPv6 address of the router's interface as a key. The elements stored in this hashtable are vectors called slps. An slps vector stores objects that describe the characteristics of an interface within it.

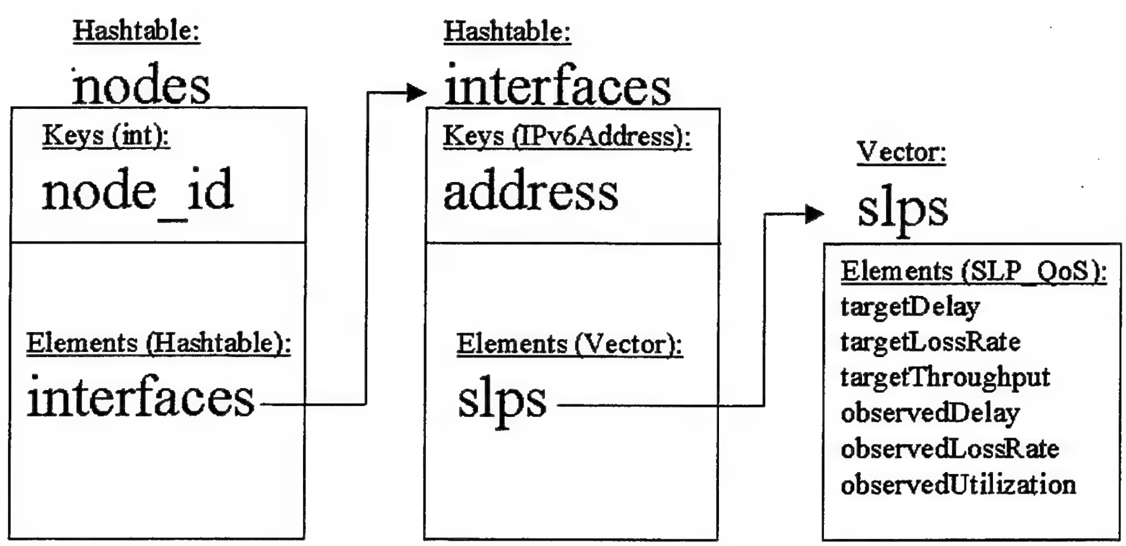


Figure 7. The Nodes Class Object

b) Paths

The Paths object is a hash table that uses the assigned path id of the router as the key (see Figure 8). The elements stored in this hash table are called path objects. A

path object contains several attributes. Among these are a hashtable maintaining information about flows that are assigned to this path. The flow id assigned to a flow is used as the key for the hashtable. The elements stored in the hashtable are Flow_QoS objects. A Flow_QoS object contains the negotiated and observed QoS parameters for the flow. Another attribute of the path object is a vector called SLPSequence. The SLPSequence vector stores an ordered list of the service level pipes that make up the path. These ServiceLevelPipe objects store the IPv6 address and service level of the service level pipe.

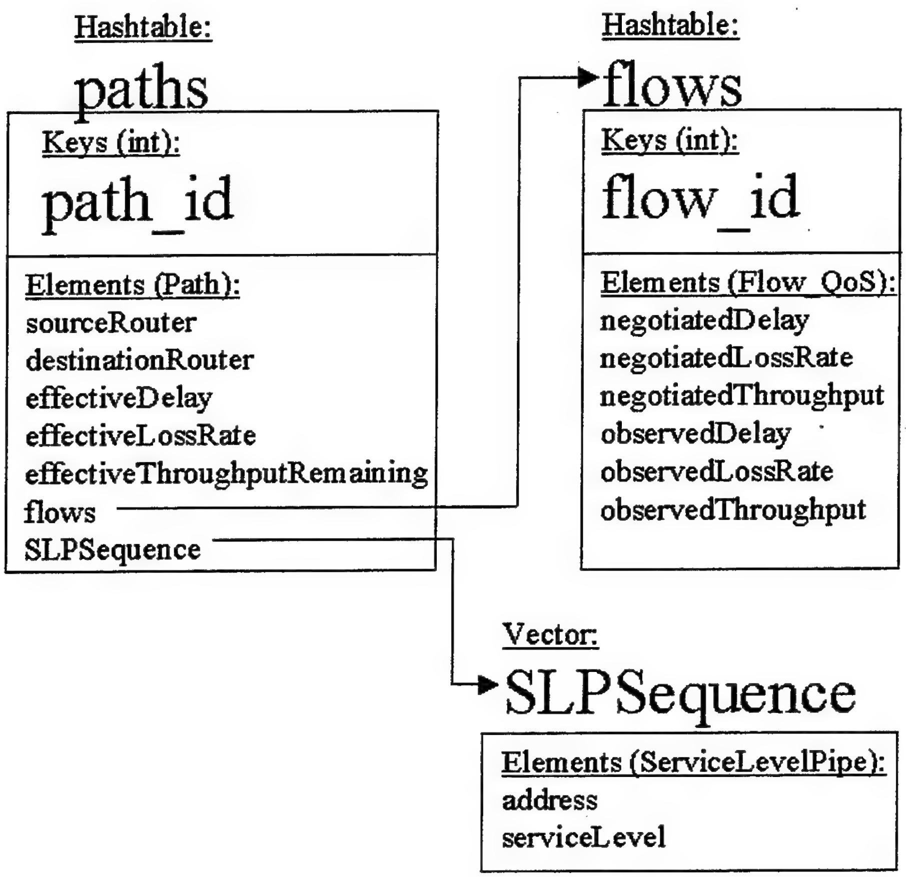


Figure 8. The Paths Class Object

c) *Links*

The links object is a hash table that uses the derived link id of a network segment as the key (see Figure 9). The elements stored in this hash table are integers that describe the bandwidth of the link.

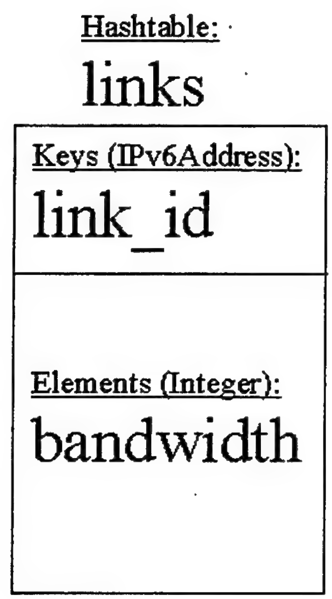


Figure 9. The Links Class Object

D. ROUTER MODEL

The router is an application that uses a layered architecture to emulate the routing of IPv6Packets within a SAAM network. Figure 10 illustrates the components of a SAAM router and the protocol layers they are associated with. The router components will be explained in detail in the following sections. For now, it is important to understand that the router model *emulates* a protocol stack.

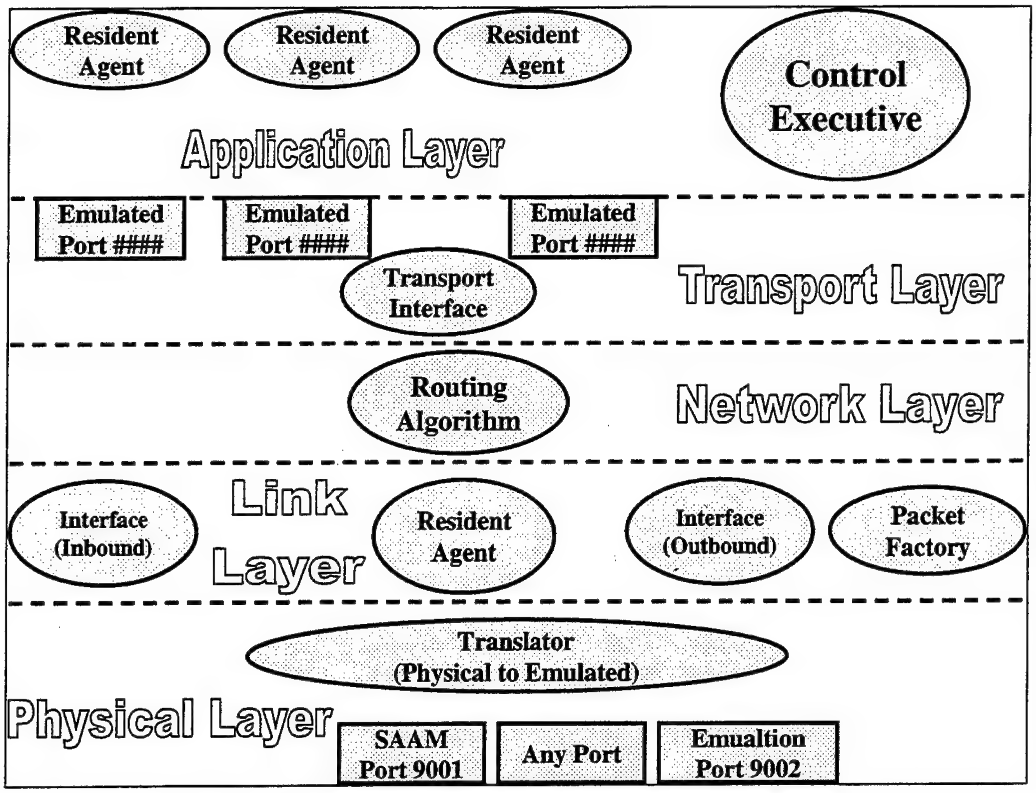


Figure 10. SAAM Router Model With Protocol Layers

The router model has been designed to operate within existing IPv4 networks; and, as such, an emulated protocol stack was necessary to enable the router to act as if it operates within an IPv6 network. The Physical Layer acts as the bridge between IPv4 and IPv6. IPv4 packets arrive on real UDP ports at this layer, whereupon the encapsulated MAC is examined and the Link Layer is notified. The Link Layer strips the MAC address, queues the packet and notifies the Network Layer. The Network Layer examines the IPv6 header and routes the packet either back down to the Link Layer or up to the Transport Layer depending on the destination IPv6 address. The Transport Layer strips the *emulated* UDP header and forwards the IPv6 packet on the appropriate *emulated* UDP port. It is important to keep the distinction between *emulated* UDP ports and *physical* UDP ports in mind when reading the sections that follow. To avoid confusion, UDP ports referred to in the Transport Layer of the emulation will be preceded with the word *emulated* in italics.

1. Dynamic Configuration Support

The router is a dynamic component within a SAAM autonomous region. It is designed to be dynamically and automatically configured by the server through the use of control traffic. This functionality enables the server to dictate the treatment of data flows within its region. In a SAAM network, the router should be required to make very few decisions. Further, for a majority of the decisions the router is required to make, the server should be capable not only of providing the algorithms required to make them, but also of replacing those algorithms when necessary. For instance, since traffic levels will vary from router to router, the server can update the scheduling algorithm of routers experiencing heavier traffic flows in order to increase the delivery rate of higher priority traffic through these routers. The routers' support for resident agents is what allows the server the flexibility to dynamically update them. In the SAAM architecture, a resident agent is a software component that is deployed to a specific router.

2. Packet Formats

Two packet structures were designed for passing information in this model. Both types are encapsulated in an IPv4 packet to enable them to be passed on current IPv4 networks. Figure 18 describes the packet structure used by the demo station to interact with the routers and servers. Figure 20 describes the packet structure used by the components within the architecture to interact with one another.

a) The Demo Packet

A demo packet is shown in Figure 11. This packet, which is sent from the demo station, need only contain a SAAM packet encapsulated within an IPv4 packet. The Translator listens for these packets on the physical UDP port designated as the emulation port. When a demo packet arrives on the emulation port, the encapsulated SAAM packet is extracted and forwarded to the Packet Factory for processing.

IPv4 packet

Physical IP		varied
IPv4	UDP 9002	Payload

SAAMPacket

9	varied
Header	Payload

Figure 11. Demo Packet Structure

b) The Emulation Packet

An Emulation packet (Figure 12) is a packet that is sent between components of the SAAM architecture. Emulation packets are structured to reflect the layers of the *emulated* protocol stack. The outer layer of the packet contains the IPv4 header and UDP information to enable the packet to travel on existing networks. The MAC field corresponds to the link layer of the *emulated* protocol stack. The *emulated* NIC strips off this field. The remaining portion of the IPv4 payload consists of an IPv6 packet. The router uses the IPv6 packet to route traffic within SAAM.

IPv4 packet

Physical IP		varied
IPv4	UDP 9002	Payload

MAC

1
Mac

IPv6Packet

40	varied
Header	Payload

UDPHeader

8
Emulated UDP Hdr

SAAMPacket

9	varied
Header	Payload

Figure 12. Emulation Packet Structure

c) *The SAAM Packet*

The SAAM packet (Figure 13) is the means by which control traffic is passed between hosts within the SAAM architecture. A SAAM packet may contain several different control traffic updates. A server, for example, sending several flow routing table entries to a router in its region would encapsulate those entries within a single SAAM packet. As indicated by Figures 11 and 12, SAAM packets are included in both packet structures. A Demo Packet (Figure 11) consists of a SAAM packet encapsulated within the IPv4 payload, whereas, an Emulation Packet (Figure 12) consists of a SAAM packet encapsulated within the payload portion of an IPv6 packet. This IPv6 packet is combined with a MAC address and encapsulated within the IPv4 payload.

A SAAM packet consists of a header and a payload.

(1) Header. The header contains two fields: the first is an eight-byte time stamp, which should should indicate the time that the packet was sent from the source. The second field is one byte and contains an integer representing the number of updates contained in the payload.

(2) Payload. The payload portion of a SAAM packet could contain one or more Server-Initiated Functions (SIFs). A SIF can be either a message or a resident agent.

SAAMPacket

9	varied
Header	Payload

SAAMHeader

8	1
TS	#of Updates

Structure of payload portion

1	1	name length	2	Bytecode length
Type	name length	name of classfile	Bytecode length	Bytecode

ResidentAgent Example

1	1	36	2	*1024
0	*36	saam.residentagent. LinkStateMonitor	1024	byte array representing LinkStateMonitor.class

Message Example

1	1	40	2	*29
0	*40	saam.message. LinkStateAdvertisement	29	Parameters for LinkStateAdvertisement constructor

*Numbers chosen arbitrarily for these examples.

Figure 13. SAAM Packet Structure

For each SIF, the payload consists of five fields. The first is a one-byte Type field. The following Type fields have been defined thus far:

Type 0 = subclass of saam.residentagent.ResidentAgent

Type 1 = subclass of saam.message.Message

The second field in the payload is a one-byte field, which represents the length (in bytes) of the field that follows. The third field is the name of the Java class object that will be instantiated by this update. The fourth field contains two bytes that represent the length (in bytes) of the field that follows. The contents of the fifth field depends on the value of the preceding Type field.

If an entry in the packet is a resident agent (Type 0), the 'bytecode' field will contain the byte array representation of the resident agent. The router uses this byte array to dynamically install resident agents. The Control Executive, receiving a resident agent update, will use the bytecode provided to instantiate the resident agent.

If the entry in the packet is a Message (Type 1), the 'bytecode' field will contain the byte array that will be used as a parameter to the constructor of that Message (i.e. the 'state' of the Message). By passing messages in this way, we enable a form of serialization of Message objects.[2]

3. Packet Flows

Figure 14 depicts the directional flow of traffic between components of the router. The wide arrows represent the flow of packets through the router, while the thin arrows represent the passing of messages between components. The three resident agents depicted at the top of the figure represent applications sent from the server that are allowed to communicate over the network by utilizing the *emulated* ports on the router. The resident agent located between the inbound and outbound Interface represents an application sent by the server that is allowed to monitor and report on traffic that passes between components within the protocol stack. The arrows coming out of and circling back to this agent represent information being gathered from the interfaces. An example

[2] The current version of the Java Developer's Kit (JDK1.2.1) offered by Sun Microsystems uses the TCP protocol to pass serialized objects. Since the current version of the SAAM emulation passes packets via UDP datagrams, serialization, as it was introduced by Sun, was not an option.

of an agent of this type would be a link state monitor whose purpose is to monitor vital statistics about the router and report them back to the server.

The SAAM Port (9001) exists to support SAAM traffic, that is, traffic that should normally flow within a SAAM network. The Emulation Port (9002), on the other hand, only exists to enable the flow of traffic that should not occur within a hardware implementation of a SAAM network. Emulation table updates and arp cache updates are examples of such traffic.

A packet entering the router is considered a *data packet* if it arrives on the SAAM port and the destination in the IPv6 header of that packet *is not* equal to the IPv6 address of one of the interfaces on that router. In this case, the packet will pass from the Translator to the inbound Interface, then to the Routing Algorithm where the outbound Interface and information about the next hop will be determined. The packet will then be forwarded to the outbound interface and on to the Translator where it will be sent to the next physical hop on any available UDP port.

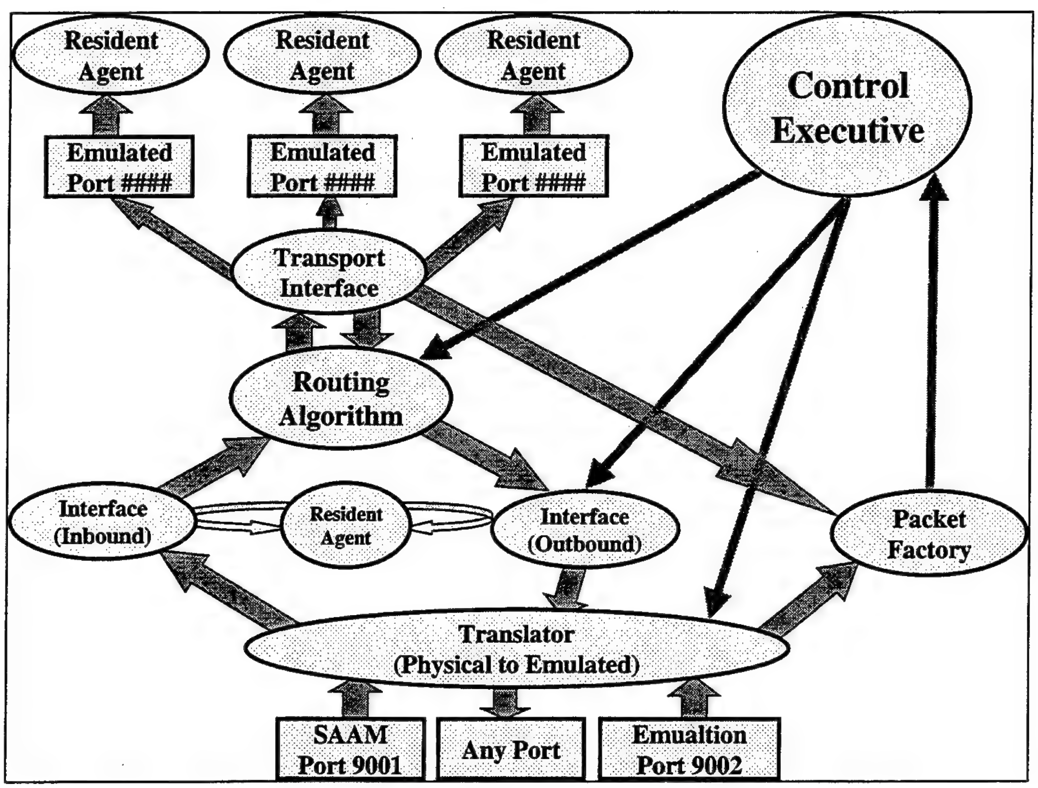


Figure 14. SAAM Router Model With Directional Traffic Flows

An inbound packet is considered to be an *application layer packet* if it arrives on the SAAM port and the destination in the IPv6 header of that packet *is* equal to the IPv6 address of one of the interfaces on that router. In this case, when the Routing Algorithm receives the packet, it is forwarded up to the Transport Interface where the *emulated* UDP header is stripped off and the destination port is determined. The Transport Interface then forwards the packet on the appropriate emulated UDP port to the agent(s) listening on that port.

An inbound packet is considered to be *control traffic* if it is an application layer packet that is destined for the *emulated* UDP port that has been designated the *emulated* SAAM Control Port. Additionally, all packets arriving on the *physical* port designated for Emulation traffic (port 9002 in this example) are to be considered control traffic. Control traffic arriving on the SAAM port is passed up to the Transport Interface where the *emulated* UDP header is stripped off. The Transport Interface then forwards the remaining SAAM packet to the Packet Factory (described below) for processing. Control traffic arriving on the Emulation port is stripped and the remaining SAAM packet is forwarded to the Packet Factory for processing.

a) The Translator (Inbound Traffic)

The Translator is not a part of the SAAM architecture. Rather, it is an aide to demonstrate the architecture's proof of concept. The purpose of the translator is to perform the conversions needed for the SAAM Router to operate in an IPv4 environment. Figure 15 illustrates the flow of traffic through the Translator.

When a packet arrives on the emulation port, the Translator simply forwards the packet to the Packet Factory for parsing. When a packet arrives on the SAAM port, the translator forwards the packet to the emulated network interface cards on the router.

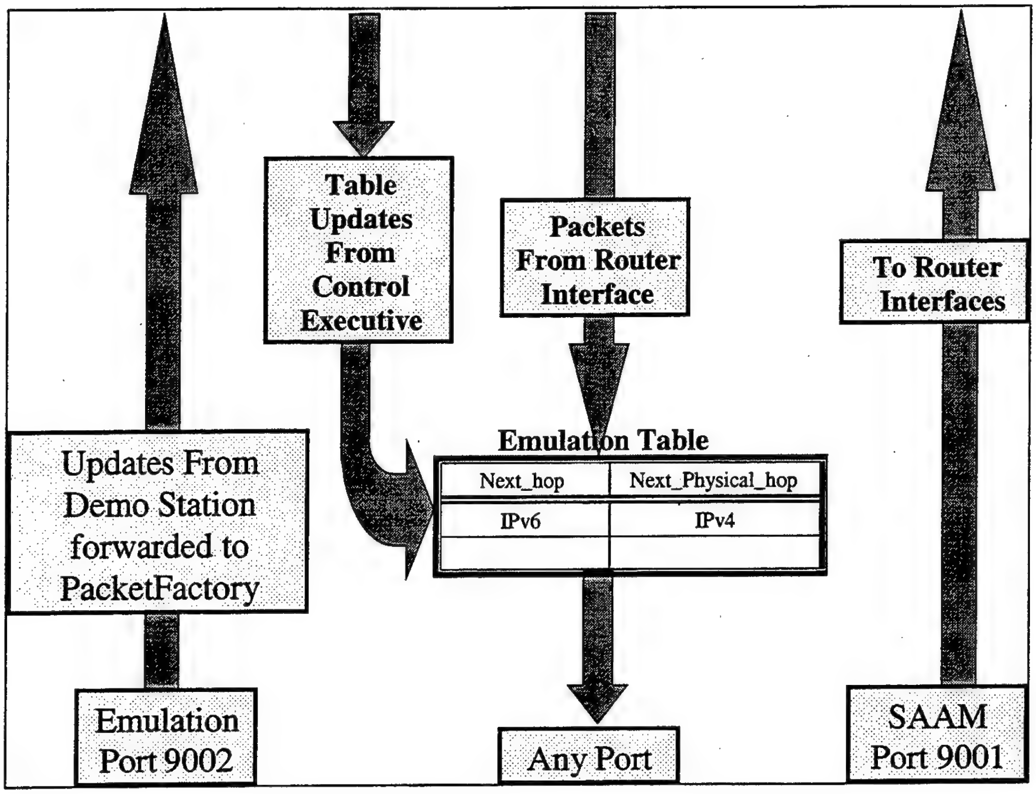


Figure 15. Translator Flow

For outbound traffic, the translator has an emulation table, which contains a mapping between the IPv6 address of the next hop and its associated IPv4 address. When an outbound packet arrives from the routing algorithm, the Translator performs an emulation table lookup, and forwards the packet out any local port to the SAAM port of the next physical hop, based on the IPv4 address retrieved.

The demo station updates the emulation table by appending an emulation table entry to a SAAM packet and forwarding the packet to the emulation port of that router. The Control Executive will receive the entry and pass it on to the Translator.

b) The Interface (Inbound Traffic)

A router Interface performs data link layer operations for the emulated router. An Interface contains an emulated Network Interface Card (NIC), one queue for inbound traffic, one queue for each service level provided by the router for outbound traffic, and a Scheduler provided by the server that dequeues packets from the service level queues and forwards them to the NIC.

Figure 16 illustrates the flow of inbound traffic through the router Interface. When the NIC within the Interface receives notification that a packet has arrived, the MAC address of the packet is compared to the MAC address of the NIC. If the MAC addresses are not equal, the NIC disregards the notification, otherwise, the MAC address is stripped off and the packet is queued until the Routing Algorithm is ready to process it.

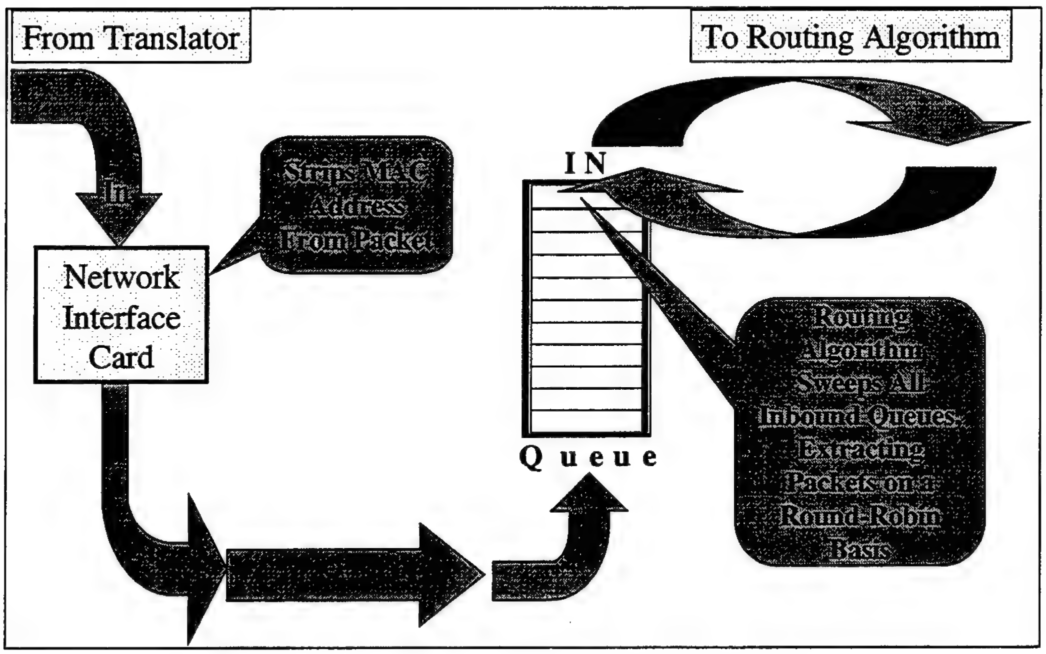


Figure 16. Interface Flow (Inbound Traffic)

c) The Routing Algorithm

The Routing Algorithm performs the network layer operations for the emulated router. IPv6 packets are routed within this component. The Routing Algorithm contains a Flow Routing Table resident agent which consists of three fields: A flow ID used as the lookup key, an IPv6 address of the next hop associated with that flow ID, and a byte representing the service level associated with that flow ID. The Routing Algorithm also contains an ARP Cache resident agent, which is a table mapping the IPv6 address of the next hop found in the flow routing table to the MAC address associated with that IPv6Address.

Figure 17 illustrates the flow of traffic through the Routing Algorithm. When an Interface places an IPv6 packet into its inbound queue, it notifies the Routing Algorithm. The Routing Algorithm then wakes up (if it is sleeping), and begins sweeping the inbound queues of each Interface, on a round-robin basis, looking for packets. When it finds a packet in one of the queues, the Routing Algorithm dequeues the packet, processes it, and moves on to the next inbound queue.

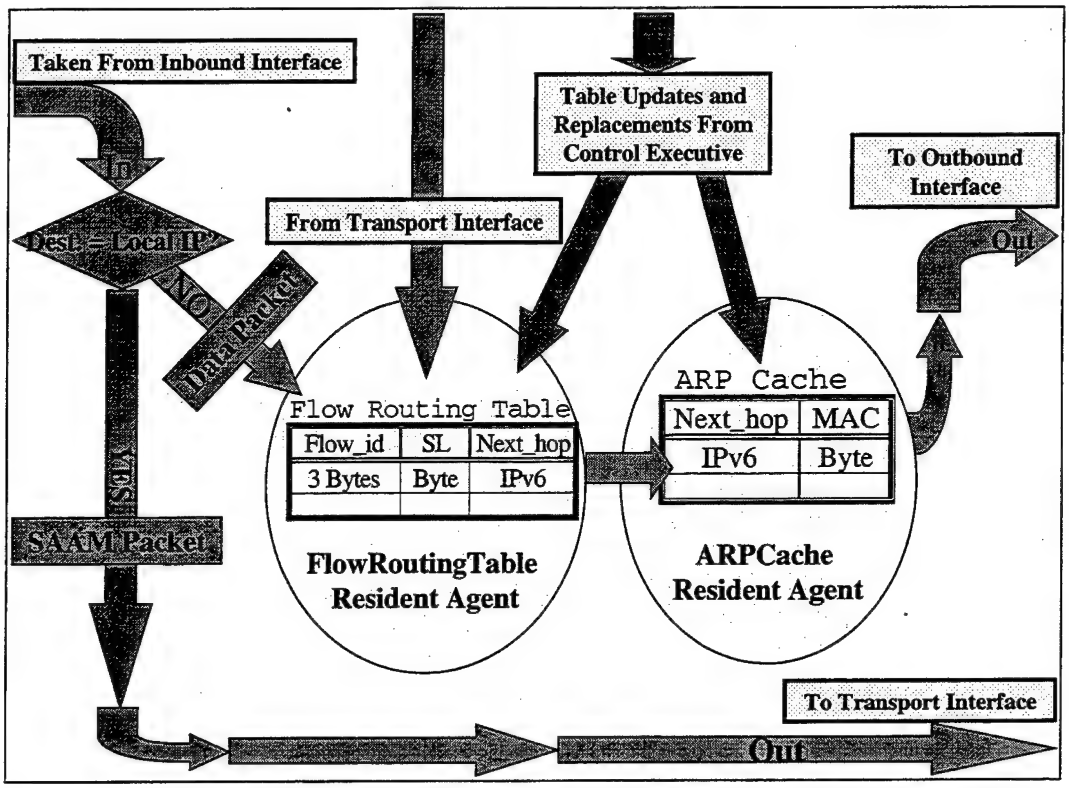


Figure 17. Routing Algorithm Flow

When a packet is dequeued by the Routing Algorithm, the destination IPv6 address contained in the packet's header is examined. If the destination IPv6 address matches the IPv6 address of one of the Interfaces on this router, the Routing Algorithm forwards the packet up to the Transport Layer. If not, the Routing Algorithm uses the three-byte flow ID contained in the packet's header as a lookup key to determine the next hop and the service level associated with the packet. Then an ARP cache lookup is performed using that next hop as a lookup key. This lookup determines the MAC address associated with the next hop.

Once the Routing Algorithm has determined all necessary information about the next hop, it compares the network portion of the IPv6 address of the next hop to the network portions of the IPv6 addresses of each Interface on the router. This comparison will determine which Interface on the router is on the same network as the next hop. The packet is then forwarded to that Interface.

If the Routing Algorithm makes one pass in which all inbound queues are empty, the Routing Algorithm goes back to sleep until it receives another notification from one of the inbound Interfaces.

d) The Interface (Outbound Traffic)

Figure 18 illustrates the flow of outbound traffic through the router Interface. Once the routing algorithm determines the outbound interface, the packet is passed to that interface where it is queued into the appropriate service level queue. The current scheduler for that interface examines the queues according to its scheduling algorithm. Since the scheduler is a resident agent, the server or demo station could easily replace this algorithm, if necessary, to improve traffic flow within the autonomous region.

The packet is then passed to the interface's emulated NIC where it is merely forwarded on to the translator.

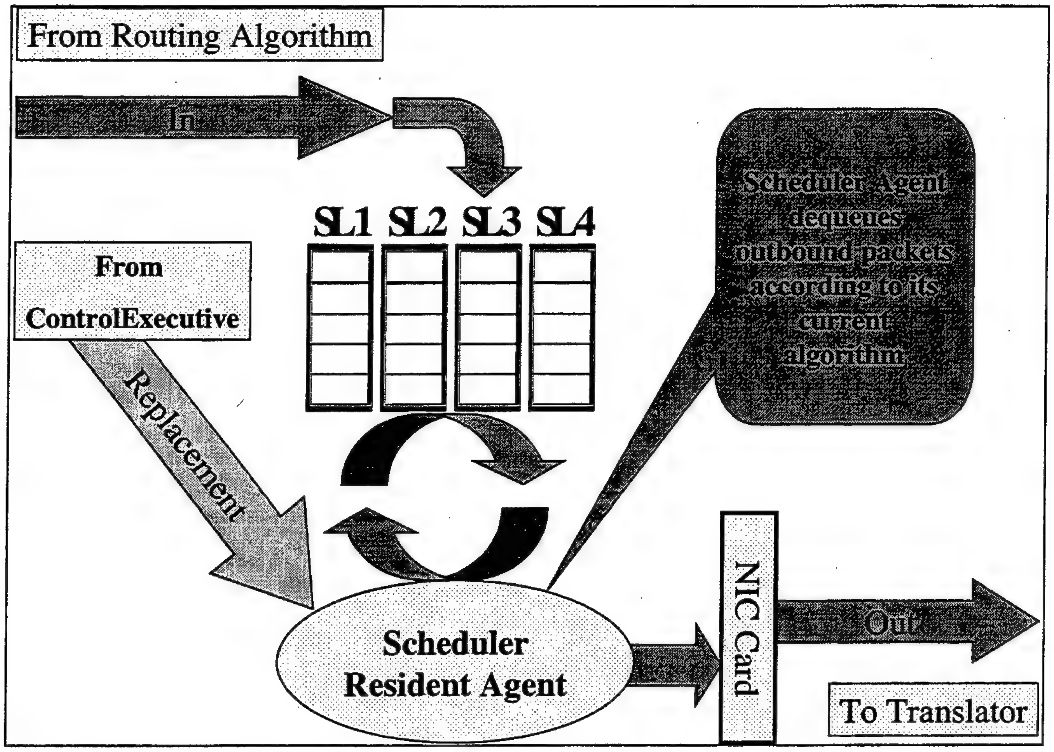


Figure 18. Interface Flow (Outbound Traffic)

e) The Transport Interface

The Transport Interface performs the transport layer operations for the emulated router. The transport layer is responsible for delivering packets to the applications for which they are destined. Figure 19 illustrates how the Transport Interface as the packet-delivery mechanism within the emulated router.

First, the emulated UDP header is stripped from the packet. The *emulated* destination port from that header is used to determine which *emulated* port the packet is destined for. If the packet is destined for the *emulated* port that was designated as the *emulated* SAAM control port, the packet is forwarded to the Packet Factory. Otherwise, the Transport interface delivers the packet on the *emulated* port whose id was extracted from the UDP header.

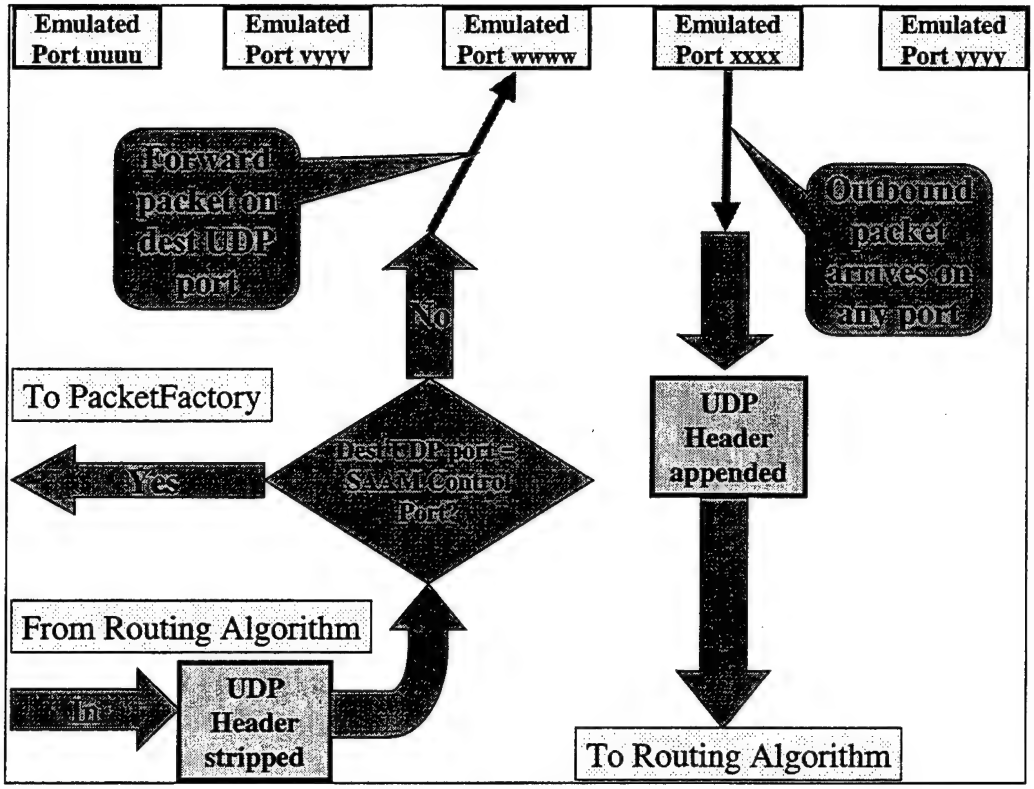


Figure 19. Transport Interface Flow

When the Transport Interface receives an outbound packet, it appends a UDP header to the packet and forwards it to the Routing Algorithm.

f) The Packet Factory

The Packet Factory is capable of assembling and disassembling SAAM packets. The router uses a Packet Factory to disassemble incoming packets that are forwarded from either the Translator or the Transport Interface. Figure 20 illustrates what happens when an incoming SAAM packet is delivered to the Transport Interface.

When the packet arrives, the Packet Factory strips off the header to determine the number of updates contained within the packet (see Figures 11 & 12 for packet structure), then it begins parsing the payload to extract the individual router updates. As each update is extracted, the Packet Factory forwards that update to the Control Executive for analysis and processing.

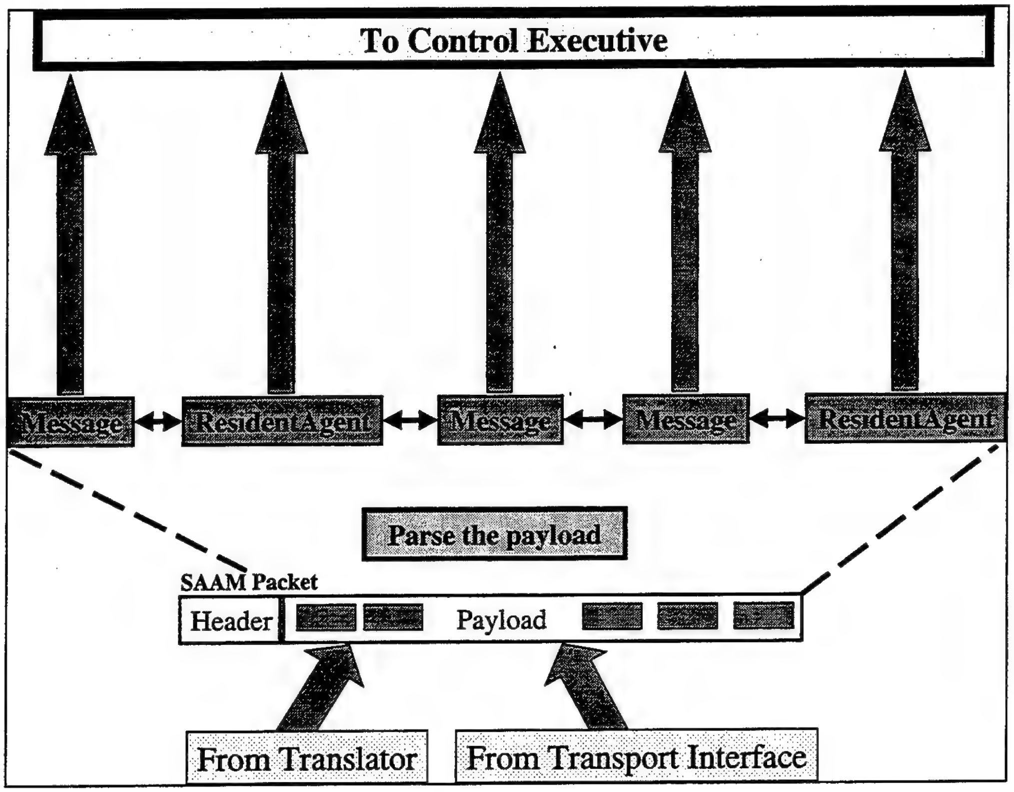


Figure 20. Packet Factory Flow

The router uses a Packet Factory to send SAAM packets to other routers or to the server. The Packet Factory builds a packet by appending the individual updates to the packet, and then appending a header when the packet is ready to be sent.

g) *The Control Executive*

The Control Executive is the control authority for the router. Before resident agents are installed on the router, the Control Executive analyzes them to determine whether or not they are safe agents. If an agent is determined to be safe, the Control Executive creates an instance of the agent. Once instantiated, agents must register with the Control Executive in order to monitor the existing components of the router or to communicate across the network. Figure 21 illustrates how the Control Executive processes messages and resident agents.

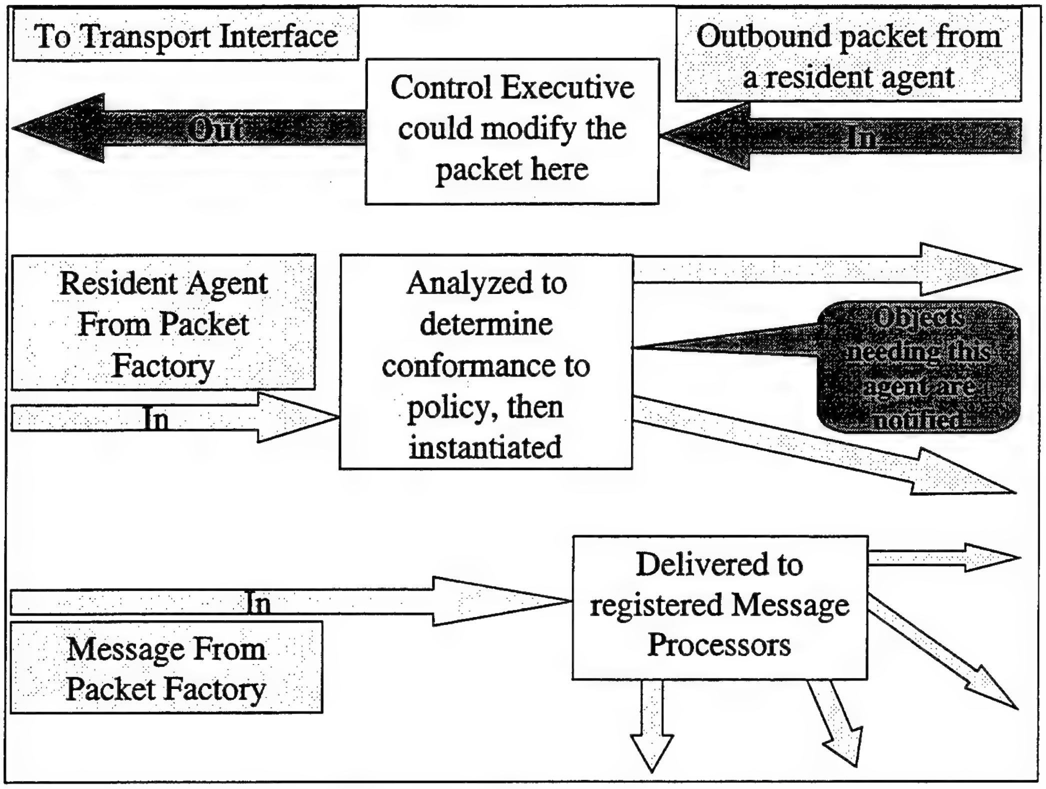


Figure 21. Control Executive Flow

The Control Executive acts as the portal for applications that wish to communicate over the network. Applications send flow requests, receive flow responses, and send outbound traffic through the Control Executive. Inbound packets destined for applications are forwarded directly from the Transport Interface to the applications on the *emulated* port that the applications are listening to without intervention from the Control Executive.

V. PROTOTYPING

Prototyping is the implementation of a model, often in a very simplified form. For this thesis, we chose to focus on the implementation of the critical functions of a SAAM server and router. In order to allow our prototype to function in the context of an IPv6 environment, an emulated environment was created. On top of this emulated environment, we used a demo station to configure routers. That is, this demo station passes the parameters needed to define the initial configuration of a router. The demo station also is used to initialize a server on a specified router. This chapter describes the organization of the server and router prototype.

A. SERVER PROTOTYPE

The functionality of a server can be divided into a small number of objects. The `Server` object itself performs the primary functions. This `Server` object instantiates a `PathInformationBase` object. A `PathInformationBase` object can be instantiated in one of two ways. The first, a `DatabaseStructure` object, uses a relational database to store the path information. The second, a `ClassObjectStructure` object, maintains PIB data in volatile memory. The final major object is the server's console, which is a Java Swing application that provides the administrator with an interface that can be used to manage the server.

The `Server` and `PathInformationBase` objects use the other objects described below to exchange data. The `SLP` (service level pipe) object describes the QoS characteristics of a service level pipe. The `SLPSequence` object describes the sequence of `SLP` objects that define a particular path.

1. Server Class

The Server is the object within the SAAM architecture that maintains a view of the network. This view is used to assign flows to paths. The Server object exchanges messages with the routers assigned to its region. It is instantiated by the Translator object. The Server contains a number of class variables. First, a PathInformationBase object, named PIB, is instantiated in the constructor. The type of the PIB instantiated is based on the value of a string parameter, either "database" or "classobject". The PIB contains information about the network. Second, a ControlExecutive object, named controlExec, is passed to the Server in the constructor. The controlExec object enables the Server to receive and send particular types of messages.

The next two variables described play an important role in determining the content of the PIB. An integer, Hmax, specifies the maximum number of hops that a paths may contain. Varying Hmax can greatly change the number of possible paths through the network. Another integer, numOfServiceLevels, indicates the number of service level pipes per interface in this server's SAAM region. This variable is assigned a value within the processHello() function. Each interface in a SAAM region has the same number of service levels. The string, serverIPv6, contains the IPv6 address of this server.

Two variables are used to limit the frequency with which paths are built within the PIB. The first, a long integer named timeOfLastPIBBuild, is initialized by System.currentTimeMillis(). The next, another long integer named timeBetweenPIBBuilds, is currently arbitrarily set to 2 minutes.

a) void Server(String type, ControlExecutive controlExec)

This method constructs a Server. The string, type, specifies whether the PIB is structured as a database or a class object. The ControlExecutive, controlExec, manages the interface to the IPv6 protocol stack ensuring messages flow to and from the network. The final step taken is to reinitialize the data structures within the PIB.

b) void processHello(Hello hello)

This method receives `Hello` messages from routers and then processes them. It starts building a vector of `IPv6Address` objects from the interfaces included in the `Hello` message. This vector is passed to the PIB's `doesRouterExist()` which determines whether a router with any of these interfaces is already in the `PIB`. If this is a new router, a unique node id is assigned to it.

For each of the interfaces identified in the `Hello` message, if this interface is not known to the `PIB`, `processHello()` checks to see if the corresponding link is known to the `PIB`. If the link is not known to the `PIB`, it is added. Next, a new interface between the node and link is added. Finally, each service level pipe that is assigned within this SAAM region is added.

The next step is to build the paths that have now become available across the network because of the newly identified node and/or interfaces.

Finally, a flow request for communicating back to this node is issued. This is only possible if the PIB's `determineAllPossiblePaths()` has been executed after the processing of this particular hello message, i.e., if this is a new router. After all paths to each known router are found, this method completes with a call to `determineEffectiveQoSForPaths()`. The call to `determineEffectiveQoSForPaths()` ensures that if no QoS parameters are known about these new parts of the network, defualt values will be assigned. This initialization is required for the new paths since the paths must have effective delay and loss rate values that are less than the upper bound values contained in a flow request in order to be assigned. This reason is also why the paths must a bandwidth initialized to a value that is greater than the bandwidth value requested in the flow request.

c) void processLSA(LinkStateAdvertisement LSA)

This method is invoked when a link state advertisement message is received from a router. It processes the service level pipe status information contained in the `LSA` message. It begins by checking to see whether a router with the interface address described by this `LSA` is known to the `PIB`. If such a router is known to exist, the method

then checks to see whether the service level pipe described by this `LSA` is known to the `PIB`. If the service level pipe is known, then its status is updated. Otherwise, it adds a `SLP` along with its associated QoS characteristics. Finally, this method updates the effective QoS for the paths that pass over this service level pipe by calling `determineEffectiveQoSForPaths()`.

d) void processFlowRequest(FlowRequest flowRequest)

This method receives and processes flow requests from applications. It begins by finding a source and a destination router. These routers may be where the applications themselves reside, which is the most commonly expected situation in this prototype. The application could, however, reside on some host that is not registered with the PIB as a router. In this case, the appropriate source or destination router would be a router connected to the same link as the host containing the application.

The PIB is checked to determine whether there is a path that has the requested QoS available. If a satisfactory path is found, a unique flow id is associated with this path. Each router in the path is determined and a new flow routing table entry is sent to it. If no path can provide the requested level of QoS, then the flow is assigned an identifier of zero, which will be interpreted by IPv6 as best effort traffic.

Finally, a flow response is sent back to the application to inform the application of its assigned flow id. If the flow id that is return is zero, it is the application's responsibility to either lower its QoS expectation or to decide to terminate.

e) void sendFRTEUpdate(int sourceRouter, int flowId, IPv6Address nextHop, int serviceLevel)

This method sends a flow routing table entry update message to a router. This message provides the router with the information needed to forward packets based upon flow id. Once a flow routing table entry message is instantiated, the control executive's `send()` is called to send the message to the specified `sourceRouter`.

f) void sendFlowResponse(FlowRequest flow_request, int flow_id)

This method is used to send a response to the application that is requesting a flow id. This method is invoked by the `receiveFlowRequest()` method described above. Once a flow response message is instantiated and a source and destination port is defined, the control executive's `send()` is called to send it to the destination host.

g) void findAllPossiblePaths()

This method determines all of the possible paths that exist between any source and destination router in the network. This method is invoked by the `receiveHello()` method described above. The paths that are found are then recorded in the `PIB` for fast assignment of flows later.

All node ids are first retrieved from the PIB. For each service level, we build an array of parents of each node. A parent is node that is directly connected. Those directly connected nodes would have service level pipes that would need to be passed through to get to the child node in question.

This parent array is used to populate a path table. Each node id is assigned as the final destination of path and all of the different paths are then found by working out from this destination. For each of these destination nodes, a call is made to processPath() to find all the valid paths that go to this destination node. We make the call with a specified height of search of 1.

h) void processPath(Hashtable parent, int[] aPath, int heightOfSearch, int serviceLevel)

This method processes all valid paths that arrive at the destination node within some range of hops. For each parent of the node at the distance of heightOfSearch from the destination, a check is made to ensure that adding this new parent will cause no cycle. If this checks out, then that parent can be added and a new path can be assigned. The service level pipes in this new path are identified and their sequence numbers in this path are recorded to the PIB. Next, a check is made to see if the height of the search is less than the server's max search height of Hmax. If it is less, the method recursively calls itself with an incremented heightOfSearch variable.

i) void causeNoCycle(int[] aPath, int heightOfSearch, int justARouter)

This method checks to ensure that the addition of a specified new node to a specified path does not result in a cycle being created. This is accomplished by checking to see if the new node is already a member of the list of nodes in the path.

j) void determineEffectiveQoSForPaths()

This method determines what the effective QoS on each path in the PIB is. For each path, the service level pipes that compose it are retrieved. Then, for each of these service level pipes, we total up the delay and loss rate. The effective throughput remaining is determined by finding the minimum difference between the observed throughput and the target throughput of each service level pipe in the path.

k) void determineEffectiveQoSForPaths(IPv6Address address, int serviceLevel)

This is an overloaded method that determines the effective QoS for just those paths that pass over the specified service level pipe. For these paths, the effective delay and effective loss rate are calculated. The effective throughput remaining is determined by finding the minimum difference between the observed throughput and the target throughput of each service level pipe in the path.

2. PathInformationBase Abstract Class

The PathInformationBase is an abstract class within the SAAM architecture that dictates what methods need to be implemented by any class that wishes to perform the retrieval and storage activities involved with maintaining a picture of the network. All of its methods are declared abstract in order to require this.

a) void deleteAllData()

This method removes all current path data from the database. It is most commonly used during initialization of a SAAM server for a new network.

b) ***int doesRouterExist(Vector IPv6Addresses)***

This method determines whether a given router exists yet within the PIB. It takes in a vector of interface addresses contained within a Hello or LSA message. It returns the id of the node containing at least one of the interface addresses in the vector that was passed.

c) ***int getNewNodeId()***

This method finds an unassigned node id and adds it to the PIB. It is commonly used for assigning a new node_id to a previously unknown router.

d) ***boolean doesInterfaceExist(IPv6Address myIPv6Address)***

This method determines whether a given interface exists yet within the PIB. It takes in an interface address taken from a hello or LSA message and returns boolean true if the interface address already exists within the PIB.

e) ***boolean doesLinkExist(IPv6Address address)***

This method determines whether a given link exists yet within the PIB. It takes in the IPv6 address of an interface and returns boolean true if the link address already exists within the PIB.

f) ***void addLink(IPv6Address address, int max_bandwidth)***

This method adds a new link to the PIB. It takes in an IPv6 address of an interface and the maximum bandwidth of this network segment.

g) ***void addInterface(int node_id, IPv6Address address)***

This method adds a new interface to the PIB. It is commonly used when receiving a hello message. It takes in the id of the router whose interface is being added as well as the address of this interface.

h) boolean doesSLPExist(IPv6Address address, int service_level)

This method determines whether a service level pipe exists yet within the PIB. It takes in the address of its interface as well as the service level that this logical pipe is providing. It returns boolean true if this SLP is already in the PIB.

i) void updateSLP(IPv6Address address, int service_level, int delay, int loss_rate, int throughput)

This method updates the status of a known SLP's delay, loss rate and throughput. In addition to these parameters, it also requires the passing of the IPv6 address of the interface and the service level of the service level pipe that is being described.

j) void addSLP(IPv6Address address, int service_level, int target_delay, int target_loss_rate, int target_throughput)

This method add a previously unknown SLP to the PIB along with its target QoS characteristics. It takes in the IPv6 address of the interface and the service level of the service level pipe that is being described, as well as the target delay, loss rate and throughput.

k) int findARouterOnLink(IPv6Address address)

This method finds a router id that has an interface on the same link as the host making the flow request. It takes in the IPv6 address of the interface of the host requesting the flow and returns the id of a router on this link.

l) int getPathThatCanSupportFlowRequest(int source_router, int destination_router, FlowRequest myFlowRequest)

This method determines if there is a path that can support a particular flow request. It takes in the node ids of the source and destination routers as well as the host's request for the establishment of a flow. It returns the id of path that can support this request or else returns zero to indicate that no path can support this QoS.

m) int getNewFlowId(int path_id, int source_router, int destination_router, FlowRequest myFlowRequest)

This method finds an unassigned flow id and uses it to assign this new flow to the specified path. It takes in the path id and the flow request to be assigned as well as the source and destination router ids. It returns the id that is being assigned to this flow.

n) Vector getSLPSequenceOfPath(int path_id)

This method retrieves the sequence of service level pipes that make up a given path. It takes in the id of the path in question and returns a vector of service level pips that compose this path.

o) IPv6Address getInterfaceAddress(int node_id, IPv6Address link_id)

This method retrieves the IPv6 address of an interface. It takes in the id of the router whose interface is connected as well as the network address of the link that this interface connects. It returns the address of the interface that connects this node and link.

p) Vector getAllRouterIds()

This method retrieves all of the router ids assigned by the PIB so far. It returns these router ids in the form of a vector.

q) int findMaxServiceLevel()

This method retrieves the maximum service level of this SAAM region. It returns the numerically highest service level id assigned.

r) Hashtable getParents(Vector V, int service_level)

This method retrieves an array of parents for each router. A parent is a directly connected node. It takes in a vector of all router ids and the level of service for which paths are being searched. It returns a hash table of vectors containing the parents of each router.

s) int getNewPathId(int source_router, int destination_router)

This method assigns an unassigned id to a new path. It takes in a source router id and a destination router id. It returns the id for this new path.

t) IPv6Address getLinkBetween(int source_router, int destination_router)

This method identifies the network address of the physical link between two adjacent routers. It takes in the ids of the source and destination router and returns the network address of the link. If no link exists between a source and destination router, the default address of all zeros is returned.

u) void assignSLPSequence(int service_level, int source_router, IPv6Address link_id, int path_id, int sequence_number)

This method assigns a service level pipe sequence entry in the building of a path. It takes in the level of service for which paths are being built as well as the node id and link id that this service level pipe connects. It also requires the id of the path to assign and the number assigned to specify the sequence of this service level pipe in the path.

v) Vector getAllPathIds()

This method retrieves a vector of all path ids constructed by the PIB. It is most commonly used when determining the effective QoS for all paths.

w) Vector getSLPsOfPath(int path_id)

This method retrieves the service level pipes that make up a given path. It takes in the id of the path in question and returns the service level pipes that make up the path.

x) void setEffectiveQoSOfPath(int path_id, int effectiveDelay, int effectiveLossRate, int effectiveThroughputRemaining)

This method records the calculated effective quality of service parameters for a particular path. It takes in the id of the path in question along with the effective delay, loss rate and throughput remaining.

y) Vector getAllPathIdsThatTraverseSLP(IPv6Address address, int service_level)

This method retrieves a vector of all path ids that traverses the specified service level pipe. It takes in the IPv6 address of the interface and the level of service that this logical pipe is providing.

z) Vector getRouterInterfaces(int node_id)

This method retrieves a vector of all interface addresses attached to this router. It takes in the node id of the router in question.

aa) void deleteARouter(int node_id)

This method deletes a specified router from the PIB. It takes in the node id of the router to be deleted.

bb) Vector getAllLinkIds()

This method retrieves a vector of all physical link ids known to the PIB.

cc) Vector findRoutersOnLink(IPv6Address link_id)

This method retrieves a vector of all routers attached to a specified physical link. It requires the network address of the link in question to be passed to it.

3. DatabaseStructure Class

This class is a Path Information Base object within the SAAM architecture that performs SQL queries on the database containing the information needed to obtain a picture of the network for use in assigning flows to paths. Specifically, this class makes a connection to an Oracle 8i relational database. This connection is made using Sun's JDBC - ODBC bridge driver. The database is configured with a database named PIB. This PIB database is configured with tables and columns in accordance with the model. This class defines each of the methods required of a Path Information Base using SQL operations.

4. ClassObjectStructure Class

This class is a Path Information Base object within the SAAM architecture that performs operations on class objects containing the information needed to obtain a picture of the network for use in assigning flows to paths. It instantiates four hash tables objects to maintain the picture of the network: nodes, links, interfaces, and paths. These hash tables are populated with objects instantiated by four inner classes: SLP_QoS, Path, Flow_QoS, and ServiceLevelPipe. This class defines each of the methods listed above for a Path Information Base by performing searches through the appropriate data structures.

5. SLP Class

The SLP class stores the QoS parameters that describe that status of a service level pipe. This class is used as a standard way to pass this information back and forth between the server and the server's path information base.

6. SLPSequence Class

The SLPSequence class stores sequence information to identify the order of service level pipes that make up a path. This class is used as a standard way to pass this information back and forth between the server and the server's path information base.

B. ROUTER PROTOTYPE

For the router prototype, our intent was to build a basic platform using the Java programming language that would enable flow-based delivery of IPv6 packets within an emulated SAAM network. We sought to deliver a product that could be dynamically updated by messages and algorithms sent to it by its server or, for demonstration and testing purposes, by a demo station. Finally, we sought to provide a means by which

researchers and students could easily modify router characteristics or generate test traffic by creating lightweight resident agents based on our examples.

1. Emulation-Specific Messages

In addition to the server-initiated messages described earlier, it was necessary to develop several types of messages that would not normally be sent over a SAAM network. For this emulation, we decided that it would be best to allow the ARP cache and emulation tables to be populated and dynamically updated from a demo station. This gave us the flexibility we needed to easily change the network topology to demonstrate various scenarios.

Figure 22 shows the two types of messages that perform those table updates, along with a third type of message that provides initialization data necessary for standing up the interfaces on a router.

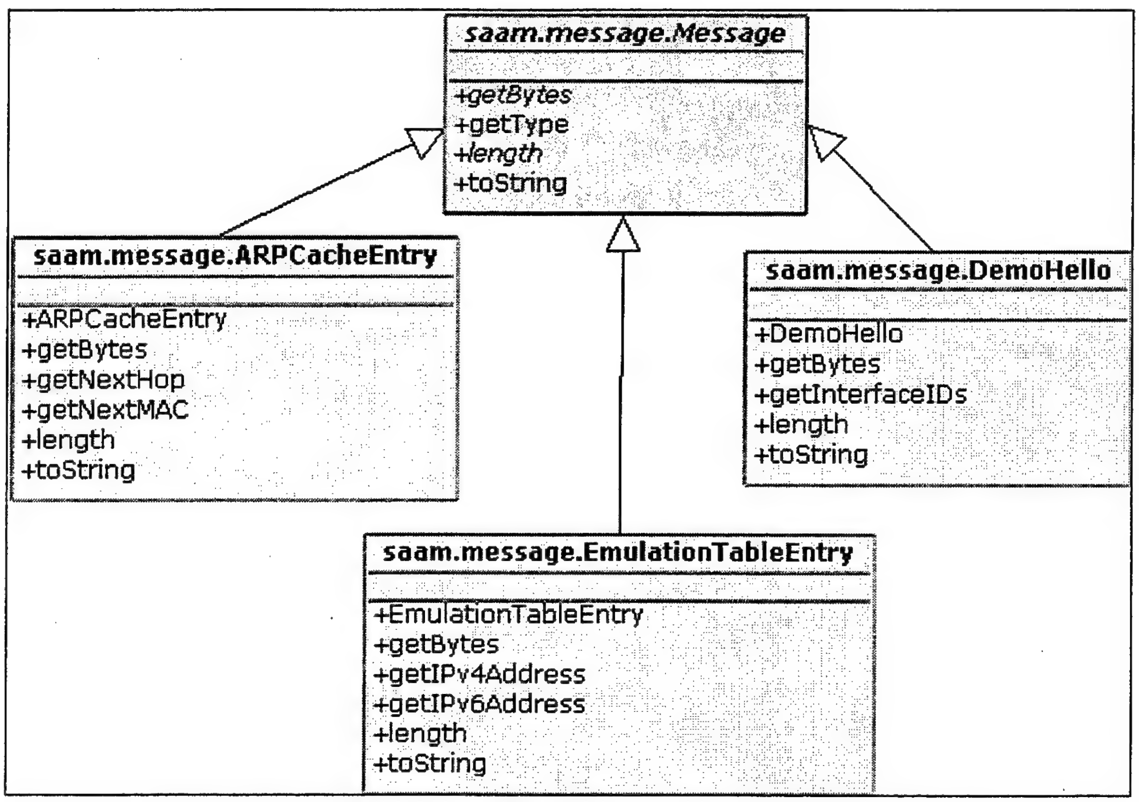


Figure 22. Messages Originating Only From the Demo Station

a) ARPCacheEntry Message

An ARPCacheEntry object contains an IPv6 address representing the next hop and a byte representing the MAC address of the next hop. The entry could be either an add or a remove depending on which constructor is used. An add is required to have both fields, whereas a remove only requires the field that is used as the lookup key in the associated table. Entries can be retrieved as a byte array using the getBytes() method.

b) EmulationTableEntry Message

An EmulationTableEntry Message contains an IPv6 address representing the next hop and an IPv4 address that maps to this IPv6 address. The entry could be either an add or a remove depending on which constructor is used. An add is required to have both fields, whereas a remove only requires the field that is used as the lookup key in the associated table. Entries can be retrieved as a byte array using the getBytes() method.

c) DemoHello Message

A DemoHello Message is simply a Message that contains a Vector of InterfaceID messages. This Message is not necessary in order to stand up a router, but it demonstrates that Message nesting can occur (encapsulating one or more messages within a single Message). To stand up several Interfaces on a router, the DemoStation can either send several InterfaceID messages, or one DemoHello Message that contains several InterfaceID messages.

2. Message Processors

A MessageProcessor is an object that is capable of processing certain types of messages. In order for a Message to be delivered on a SAAM network, there must be a MessageProcessor that is registered with the ControlExecutive to process Messages of that type. Figure 23 illustrates the objects that have implemented the MessageProcessor interface.

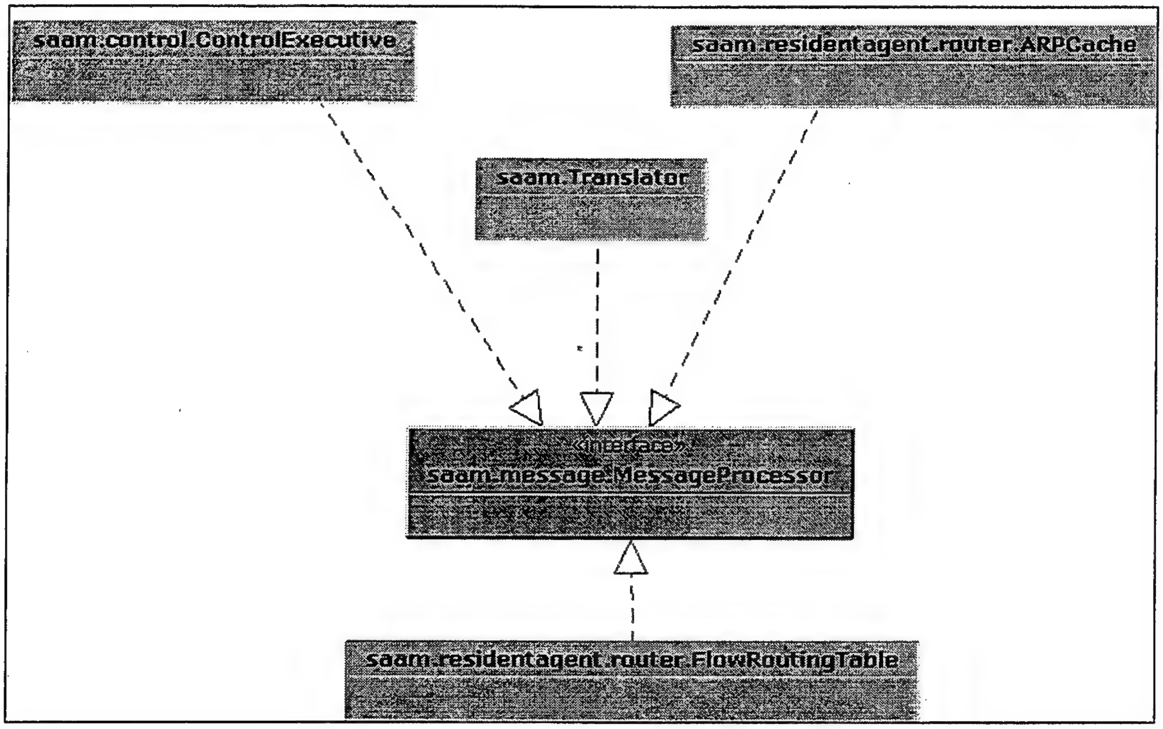


Figure 23. saam.message.MessageProcessor

To become a MessageProcessor that can actually process messages, it is not enough to simply implement this interface. Objects implementing this interface must also provide a call to the ControlExecutive's registerMessageProcessor method, passing itself as a parameter. The ControlExecutive will then make a call back to the registrant to retrieve the array of Messages this MessageProcessor would like to process. Therefor, an additional requirement of implementors of this interface is that they must supply a String array that contains the String names (fully qualified classnames) of the Messages it would like to process. The ControlExecutive will then know that when a Message of that type arrives, it should be sent to this processor.

a) ControlExecutive

The ControlExecutive acts as the registrar for all MessageProcessors. As described above, all MessageProcessors must register with the ControlExecutive to be eligible to receive messages of the type that MessageProcessor is capable of processing.

When a MessageProcessor registers with the ControlExecutive, the ControlExecutive stores that processor along with the messages the processor has registered to process.

When the ControlExecutive receives a Message, the table of MessageProcessors is consulted to determine which processor should receive the Message. Once the processor has been determined, the ControlExecutive then calls the processMessage method of that MessageProcessor, passing the inbound Message as the only parameter. The processor then proceeds to process the Message.

The ControlExecutive is itself a MessageProcessor capable of processing four types of messages.

(1) saam.message.InterfaceID. When an InterfaceID Message arrives, the ControlExecutive uses the information contained in the Message to instantiate a new Interface on the router – providing an Interface with that information has not already been instantiated.

(2) saam.message.ServerID. When a ServerID Message arrives, the IPv6Address associated with that ServerID will be set as the destination in the IPv6Header of packets sent out on flow zero[3], otherwise, the default IPv6Address will be set as the destination.

(3) saam.message.FlowResponse. FlowResponse messages are simply forwarded by the ControlExecutive to the application that sent the corresponding FlowRequest Message to the server.

(4) saam.message.DemoHello. When the ControlExecutive receives a DemoHello Message, it merely iterates through the Vector of InterfaceIDs and processes the individual InterfaceID messages in the same manner as noted in (1) above.

[3] Zero is the flow ID that has been designated in this prototype as the flow from all routers to the server.

b) Translator

The Translator is capable of processing messages of the type saam.message.EmulationTableEntry. The Translator has an Emulation Table[4] (Figure 24) that maps the IPv6 next hop of an outbound packet to its associated IPv4 address. When an EmulationTableEntry Message arrives, the record contained in the EmulationTableEntry is either added or removed from the EmulationTable.

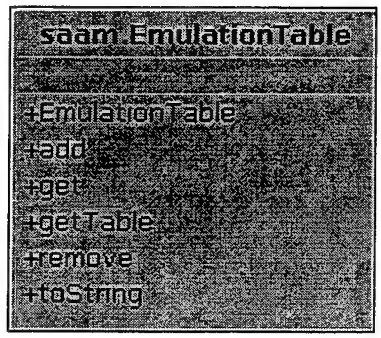


Figure 24. saam.EmulationTable

c) ARPCache

The ARPCache[5] (Figure 25) is a resident agent that, once stood up on a router, serves as a lookup table for retrieving the MAC address associated with the IPv6Address of the next hop used as the lookup key. The ARPCache resident agent is also a MessageProcessor.

[4] The EmulationTable is not a resident agent or a MessageProcessor. The Translator instantiates the EmulationTable and processes all updates to the table.
[5] Making the ARPCache a resident agent allows for the possibility of modifying the structure of the table. This may or may not be desirable. It was done here, just to show that it was possible.

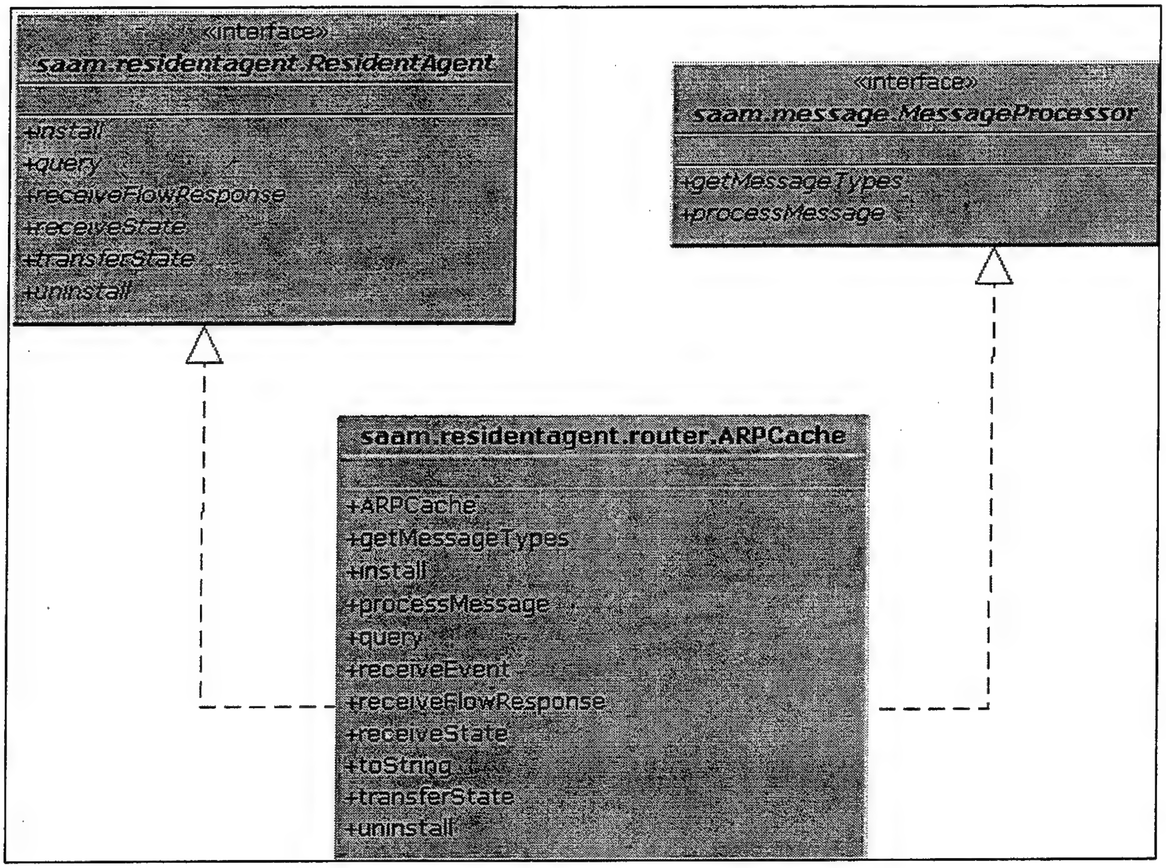


Figure 25. saam.residentagent.router.ARPCache

The ARPCache is capable of processing messages of the type saam.message.ARPCacheEntry. Based on the length of the ARPCacheEntry, the ARPCache determines whether the entry is a record that is to be added or removed. The record is then added to the ARPCache or removed from it accordingly.

d) FlowRoutingTable

Like the ARPCache, the FlowRoutingTable (Figure 26) is a resident agent that serves as a lookup table for the router. The FlowRoutingTable retrieves the service level and nextHop (IPv6Address) associated with a given flow ID used as the lookup key. The FlowRoutingTable resident agent is also a MessageProcessor. The FlowRoutingTable is capable of processing messages of the type saam.message. FlowRoutingTableEntry.

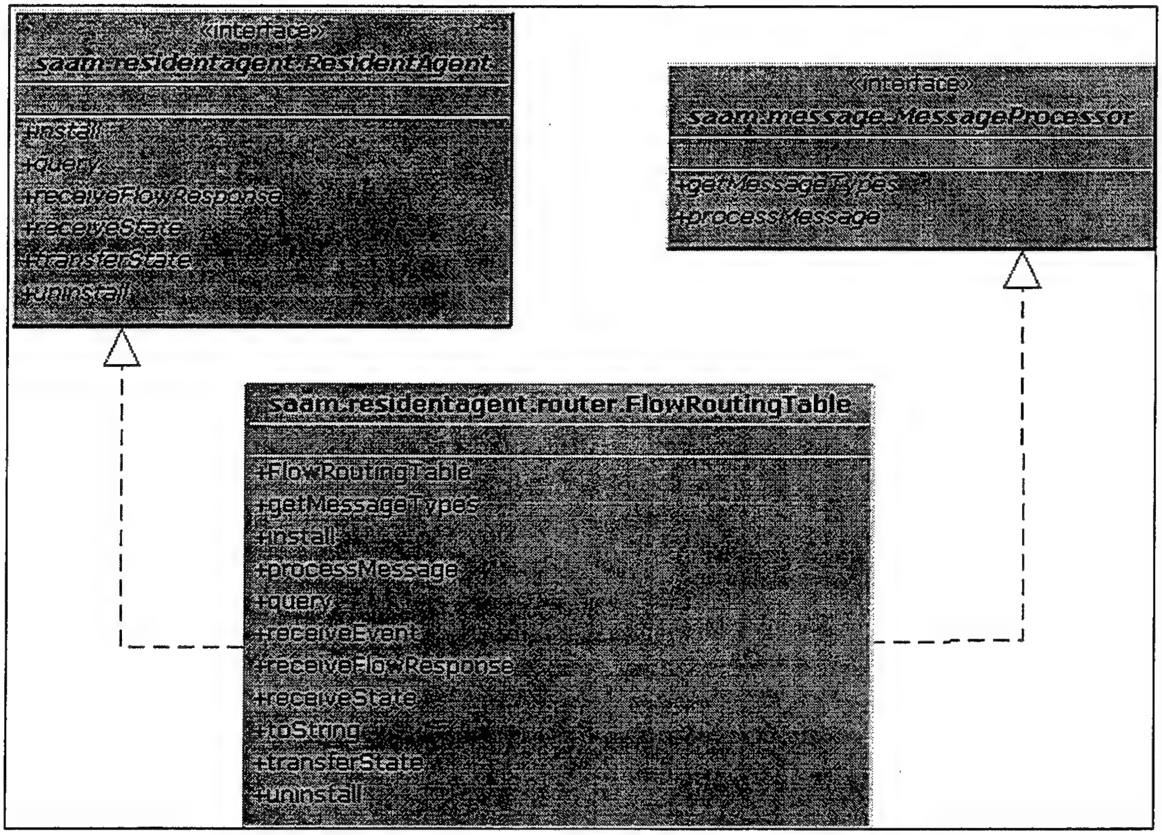


Figure 26. saam.residentagent.router.FlowRoutingTable

(1) FlowRoutingTableEntry Message Description. A FlowRoutingTableEntry object contains a three-byte integer representing the flow ID, a one-byte service level, and an IPv6 address representing the next hop. The entry could be either an add or a remove depending on which constructor is used. An add is required to have all fields, whereas a remove only requires the field that is used as the lookup key in the associated table. Entries can be retrieved as a byte array using the getBytes() method.

(2) Processing a FlowRoutingTableEntry. Based on the length of the FlowRoutingTableEntry, the FlowRoutingTable determines whether the entry is a record that is to be added or removed. The record is then added to the FlowRoutingTable or removed from it accordingly.

3. The Event Model

Objects within the router must be able to communicate with one another. Further, new objects that are dynamically introduced to the router must be allowed to communicate with the router's existing components. To enable this type of dynamically initiated inter-object communication, some sort of mechanism was necessary that would allow objects within the router to register to communicate with other objects. The following section describes the mechanism we developed to accomplish this.

a) Java's Delegation Event Model

Enabling communication between router objects is a simple task in Java. The Java programming language provides a flexible and scalable event mechanism known as the Delegation Event Model (Figure 27). In this model, event objects are passed from event sources to one or more event listeners. [14]

(1) Event Objects. An event object is simply a Java class that represents information that is to be conveyed from a source object to a destination object. Event objects can contain attributes and methods like any other Java class. The convention for naming event objects is to concatenate the desired name for the new event with the word "Event". For example, an event occurring within the SAAM environment might be named, "SaamEvent".

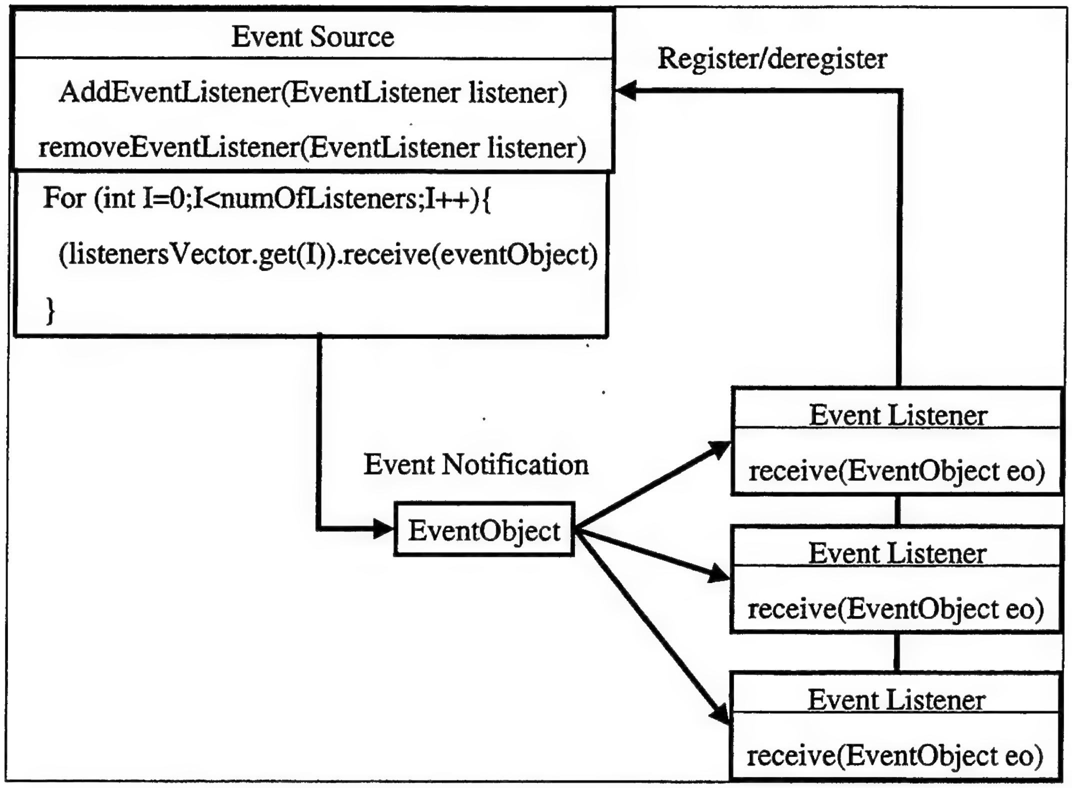


Figure 27. The Java Delegation Event Model

(2) Event Listeners. Event listeners are the destination objects will receive particular types of events when they occur. In order to receive events, listeners must register with the object from which those events originate (event source), and they must contain a method that can be called by the event source when the event occurs.

(3) Event Sources. Event sources are the objects that generate the events of interest to event listeners. Every event source should contain a private vector of listeners with accessor methods to allow listeners to be added and removed. This vector provides the mechanism for listener registration. When an event that is being listened for occurs within an event source, the event source first creates an event object that represents the state of the event. The source then iterates through its vector of

listeners and calls the method within each listener that is designated to receive the event object, passing the event object as the only parameter[6].

b) *Delegation Event Model Limitations*

In developing the router, it became apparent that the Delegation Event Model does not address certain issues related to inter-object communication that are of concern to SAAM.

(1) Order of Instantiation. In the Delegation Event Model, the order of instantiation of objects is important. For Instance, an event listener cannot register with an event source that has not been instantiated. In our attempts to conform to the model, we found ourselves designing constructors with many parameters just to accommodate message passing.

(2) Event Source Replacement. On a similar note, if one event source is to take over the notification of a certain type of event, listeners for that event must de-register with the old source and then register with the new source. This process involves a great deal of object passing and quickly becomes combersome.

c) *The Channel*

In order to overcome the limitations of the Delegation Event Model noted above, we developed the concept of a Java communication *channel*.

(1) Concept Definition. A channel is mechanism for delivering events from one or more event sources to one or more event listeners (Figure 28). A channel is a Java class object, and is thus allowed to have attributes and methods that constitute the state of the channel. A channel contains a vector of listeners and a vector of *talkers*. Under this model, event listeners become listeners to a channel. They register with a channel as they would register with any event source. The key difference here is that event sources now must register as talkers on the channel. To send an event

[6] To ensure that each listener contains this method, it is common to create an interface class that can be implemented by objects desiring to listen for particular events. For example, objects desiring to listen for events of type SaamEvent might implement an interface named SaamListener.

notification, a registered talker merely calls the talk method of the desired channel. This wakes up the channel's thread, which notifies all listeners on that channel.

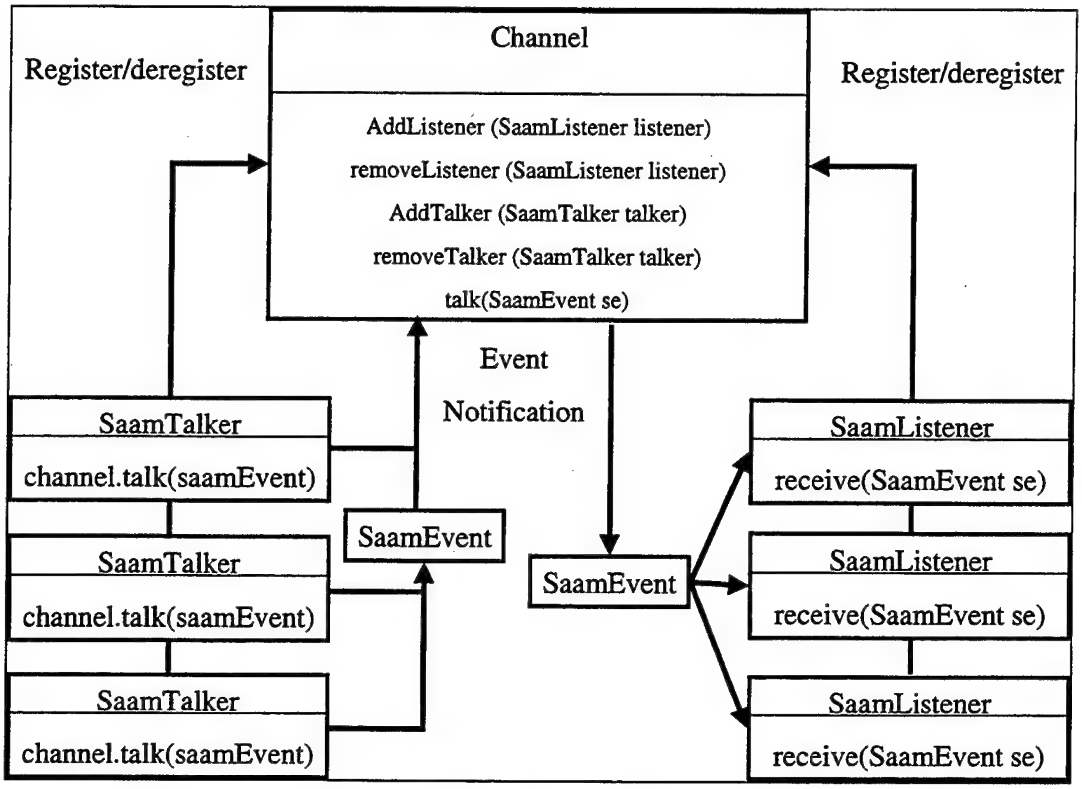


Figure 28. The Channel Concept

(2) Overcoming Delegation Model Shortcomings. Channels offer several benefits over the standard Delegation Event Model described previously. In the channel registration process, the order in which objects register is not important. Listeners are allowed to register with a channel that has no talkers and vice versa. If a listener registers with a channel that has no talkers, the listener will not receive any events on that channel until a talker registers with, and then talks on the channel. If a talker registers on a channel that has no listeners, the events sent by the talker will be dropped.

(3) Additional Benefits Offered by Channels. When we attempted to design a mechanism by which objects were allowed to communicate with an emulated UDP

port, we found that the channel concept was perfectly suited for this purpose. Much like ports, channels deliver information from source to destination in parallel through event notifications. UDP ports are accessed by a unique 2-byte integer. By assigning an integer field to the channel that represents the channel's id, we can map channel ids to UDP port numbers. All that remains is to assign an object to manage the ports.

d) Channel Management

Channels operating independently offer several benefits not previously realizable, but the true power of this concept lies with the ability to manage a group of channels. Figure 29 illustrates how an object can harness a group of channels and provide a means of controlling the important aspects of those channels.

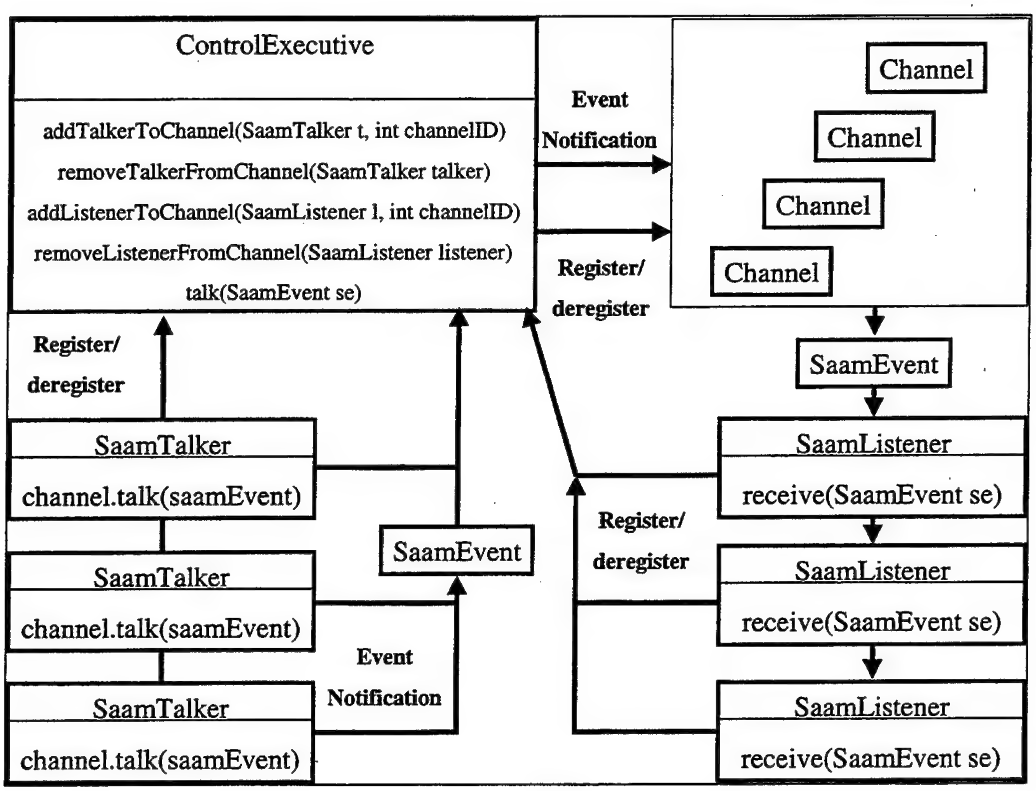


Figure 29. The Saam Router Event Model (Channel Management)

(1) The Control Executive as Channel Manager. In this model, a Control Executive acts as the registrar for a group of channels, the range of UDP ports available to an application for example. By allowing talkers and listeners to register with the Control

Executive to communicate over a channel, we can provide these objects with a means for identifying and plugging in to channels that are provided for various purposes, such as for passing packets through a router. Further, by making the channel class accessible only to objects within the control package, we provide a mechanism for controlling the monitoring and delivery of traffic on the channels we set up.

The Control Executive in our model plays the role of what Berg and Fritzinger refer to as an event adapter (Figure 30). According to them,

> An event adapter is simply an object that sits between an event source and an event listener. It interposes between the two by listening for events from the source and then may modify the event in some manner before delivering the event to the event listener.

Applications desiring to monitor an *emulated* port, to monitor channels within the protocol stack, or to send traffic on either of these must register with the Control Executive. From this position, the Control Executive can control all traffic sending and receiving that occurs on the channels.

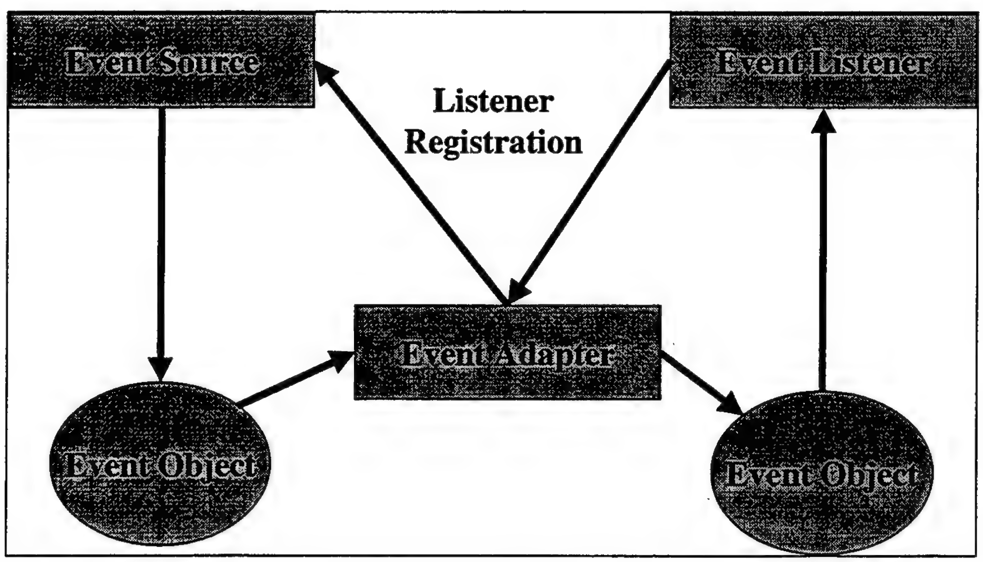


Figure 30. Event Adapter

4. Emulated Protocol Stack Components

In order to demonstrate the functionality of a SAAM router within an IPv6 network, it was necessary to develop an emulated protocol stack. The key components that make up the emulated protocol stack are highlighted in Figure 31 and described individually in this section.

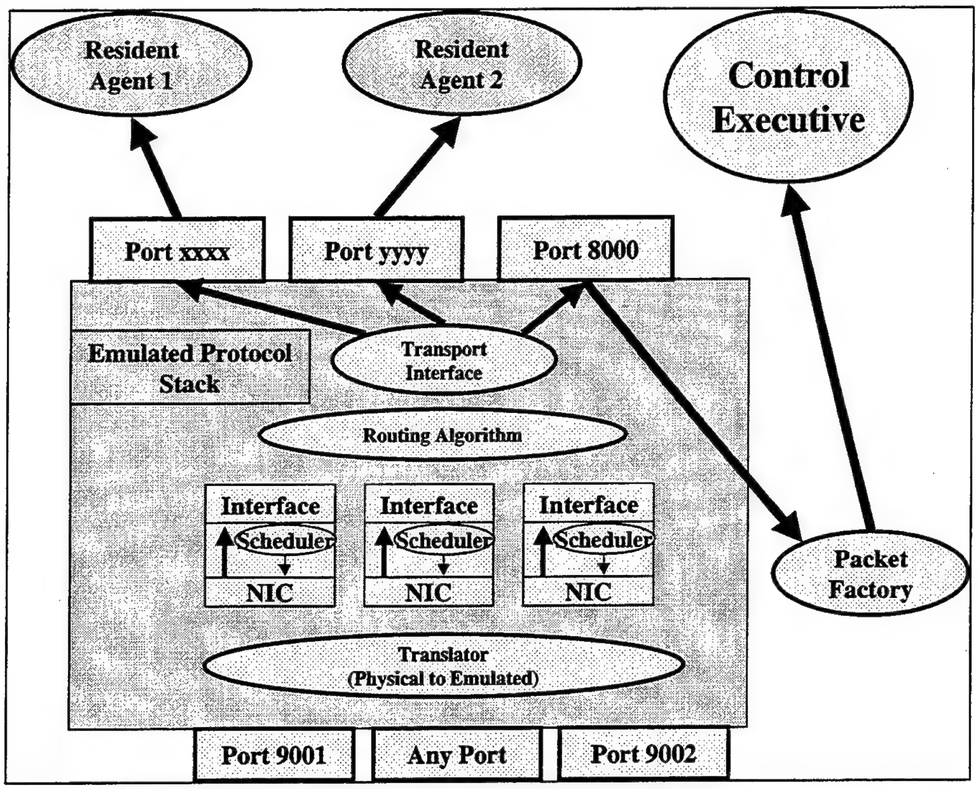


Figure 31. Emulated Protocol Stack

a) The Translator

The Translator (Figure 32) is not a part of the SAAM architecture. Rather, it is an aide to demonstrate the architecture's proof of concept. The purpose of the Translator is to perform the conversions needed for the SAAM Router to operate in an IPv4 environment.

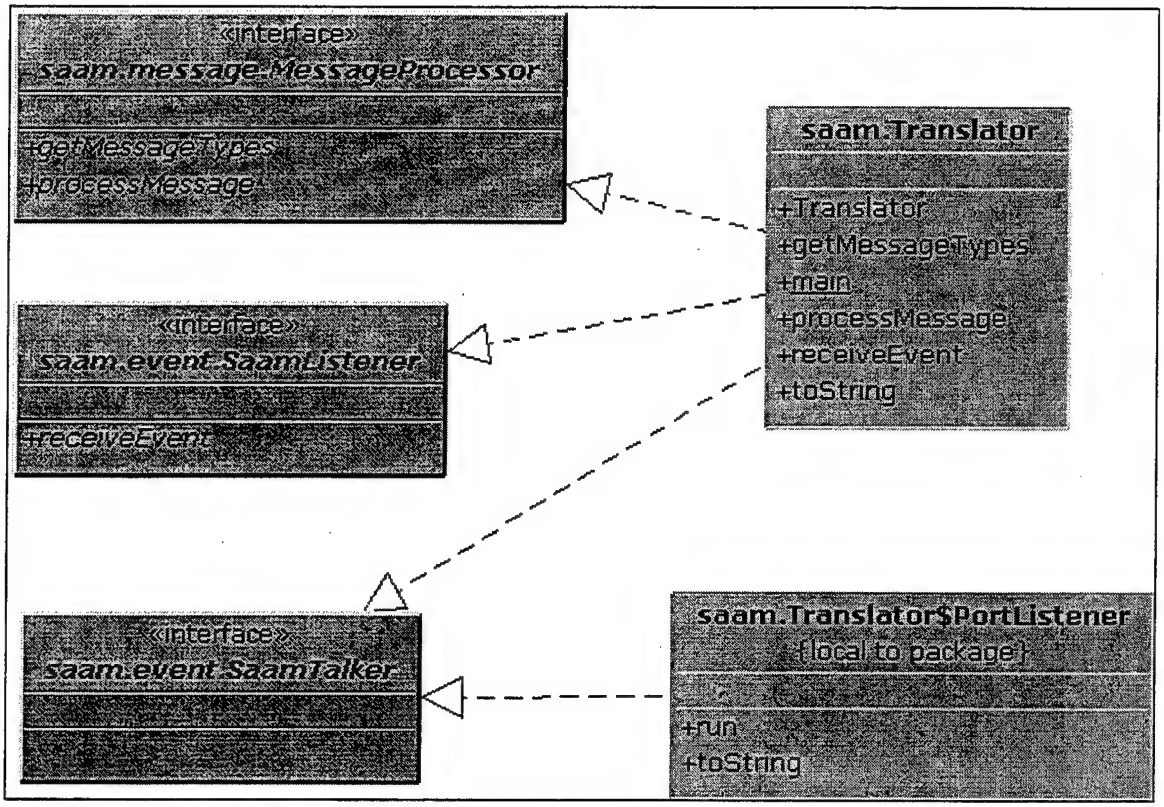


Figure 32. saam.Translator

(1) Inbound Traffic. When instantiated, the Translator first stands up a ControlExecutive and then a PortListener. The PortListener is a threaded inner class used by the Translator to receive and forward Datagram packets on the Channel designated in its constructor. When packets are received on the emulation port, the PortListener forwards them to the PacketFactory for processing.

Additionally, the Translator registers to listen on the ROUTER_STATUS_CHANNEL, which is used by the ControlExecutive to notify the Translator that all elements necessary to stand up a router have been received. Once the Translator receives this notification from the ControlExecutive, the Translator stands up another PortListener on the SAAM port (9001). When packets are received on the SAAM port, the PortListener forwards them to the NetworkInterfaceCard objects on the FROM_TRANSLATOR_TO_NICS_CHANNEL.

(2) Outbound Traffic. The Translator listens for outbound traffic on the FROM_NICS_TO_TRANSLATOR_CHANNEL. Packets arrive on this

channel in the form of a ProtocolStackEvent[7]. When the Translator receives an outbound packet, the EmulationTable is consulted to determine the IPv4 address of the next hop, then the packet is forwarded to that location.

b) The Interface

The router Interface (Figure 33) class implements both the SaamTalker and SaamListener interfaces. In order for packets to pass through the router Interface, the Interface must listen for packets coming into it, and it must talk on channels to forward the packets out. When an Interface is instantiated, it registers to communicate on several channels.

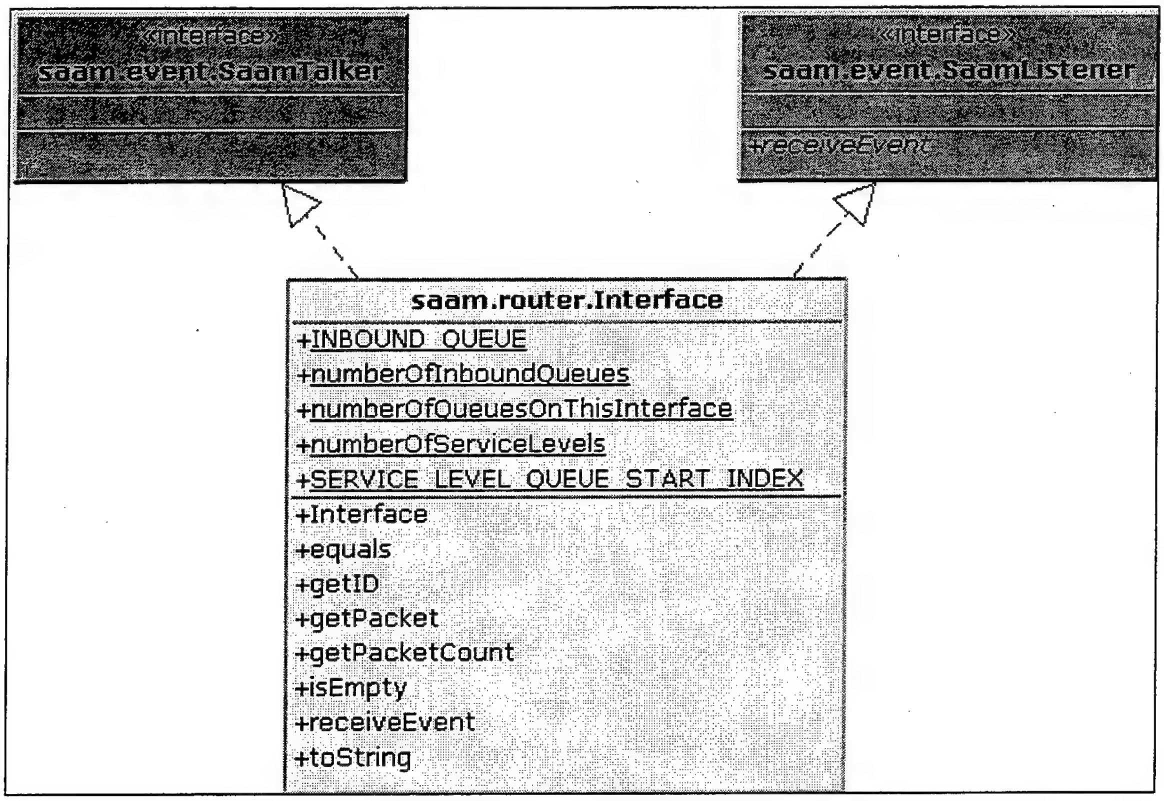


Figure 33. saam.router.Interface

(1) Inbound Traffic.

[7] A ProtocolStackEvent is an event that is fired from within the protocol stack. Normally this class is used as a means of communications between Objects within the protocol stack, such as to pass a packet from a NetworkInterfaceCard to the Translator.

(a) Channels

To receive packets from its NetworkInterfaceCard (Figure 31), the Interface registers to listen on the appropriate Channel by calling the getFromNICToInterfaceChannel method in the ProtocolStackEvent class. To notify the RoutingAlgorithm that a packet is about to be placed in the inbound queue, the Interface registers to talk on the appropriate Channel by calling the getEnqueuingInboundPacketChannel method in the ProtocolStackEvent class.

(b) Packet Processing

An Interface uses the NetworkInterfaceCard that it instantiates to strip the MAC address from the packet, then the Interface places the packet into its inbound queue where it will be extracted by the RoutingAlgorithm.

(2) Outbound Traffic.

(a) Channels

To receive packets from the RoutingAlgorithm, the Interface registers to listen on the appropriate Channel by calling the getFromRoutingAlgorithmToInterfaceChannel method in the ProtocolStackEvent class. To notify the Scheduler that a packet is about to be placed in a service level queue, the Interface registers to talk on the appropriate Channel by calling the getFromInterfaceToSLQueueChannel method in the ProtocolStackEvent class.

(b) Packet Processing

When an Interface receives outbound packets from the RoutingAlgorithm, the service level is determined and the packet is placed into the appropriate service level queue where it will be extracted by the Scheduler assigned to that Interface.

c) The NetworkInterfaceCard

Every router Interface instantiates a NetworkInterfaceCard (Figure 34) to perform link layer processing for incoming packets bound for that Interface. Like the router Interface, the NetworkInterfaceCard class implements both the SaamTalker and

SaamListener interfaces. When a NetworkInterfaceCard is instantiated, it registers to communicate on several channels.

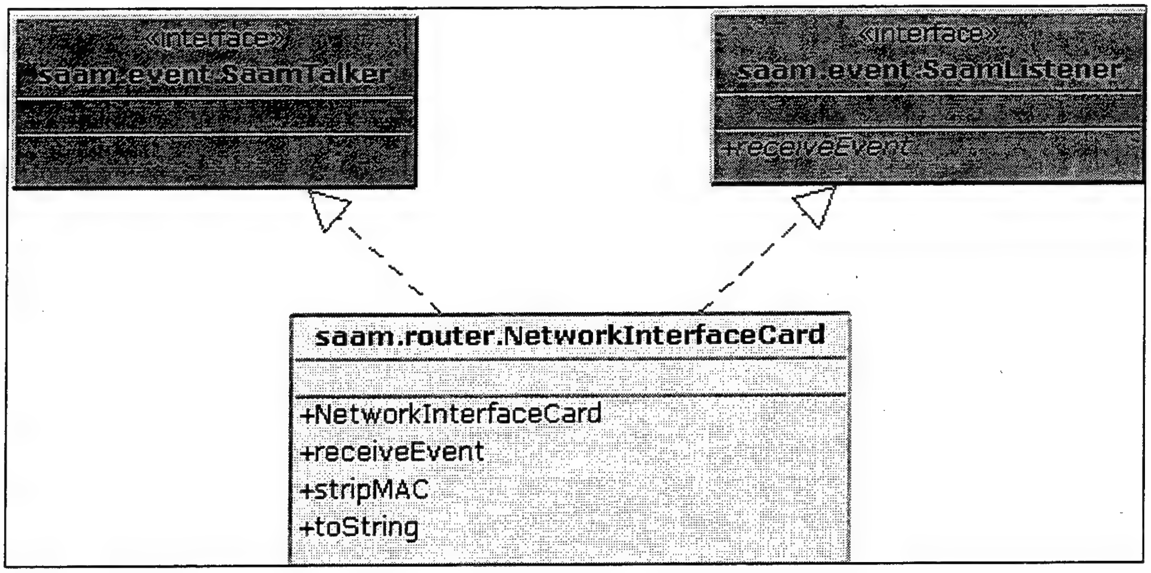


Figure 34. saam.router.NetworkInterfaceCard

(1) Inbound Traffic.

(a) Channels

To receive packets from the Translator, the NetworkInterfaceCard registers to listen on the FROM_TRANSLATOR_TO_NICS_CHANNEL defined in the ProtocolStackEvent class.

To forward the packet to the router Interface, NetworkInterfaceCard registers to talk on the appropriate Channel by calling the getFromNICToInterfaceChannel method in the ProtocolStackEvent class.

(b) Packet Processing

A NetworkInterfaceCard strips the MAC address from the packet, then forwards the packet to the Interface that instantiated that NetworkInterfaceCard.

(2) Outbound Traffic.

(a) Channels

To receive outbound packets from the Scheduler assigned to its router Interface, the NetworkInterfaceCard registers to listen on the appropriate Channel by calling the getFromSchedulerToNICChannel method in the ProtocolStackEvent class.

To forward outbound packets on to the Translator, the NetworkInterfaceCard registers to talk on the FROM_NICS_TO _TRANSLATOR _CHANNEL defined in the ProtocolStackEvent class.

(b) Packet Processing

When an NetworkInterfaceCard receives outbound packets from the Scheduler, the packet is merely forwarded on to the Translator. No additional processing is necessary because the RoutingAlgorithm has already appended the outbound MAC to the packet.

d) The RoutingAlgorithm

There is one RoutingAlgorithm (Figure 35) for every router which performs the network layer functions for the router. The RoutingAlogrithm contains the logic that determines where to forward packets when they arrive and when they are being sent out. In addition to implementing the SaamTalker and SaamListener interfaces, the RoutingAlgorithm class implements the Runnable and ResidentAgentCustomer interfaces.

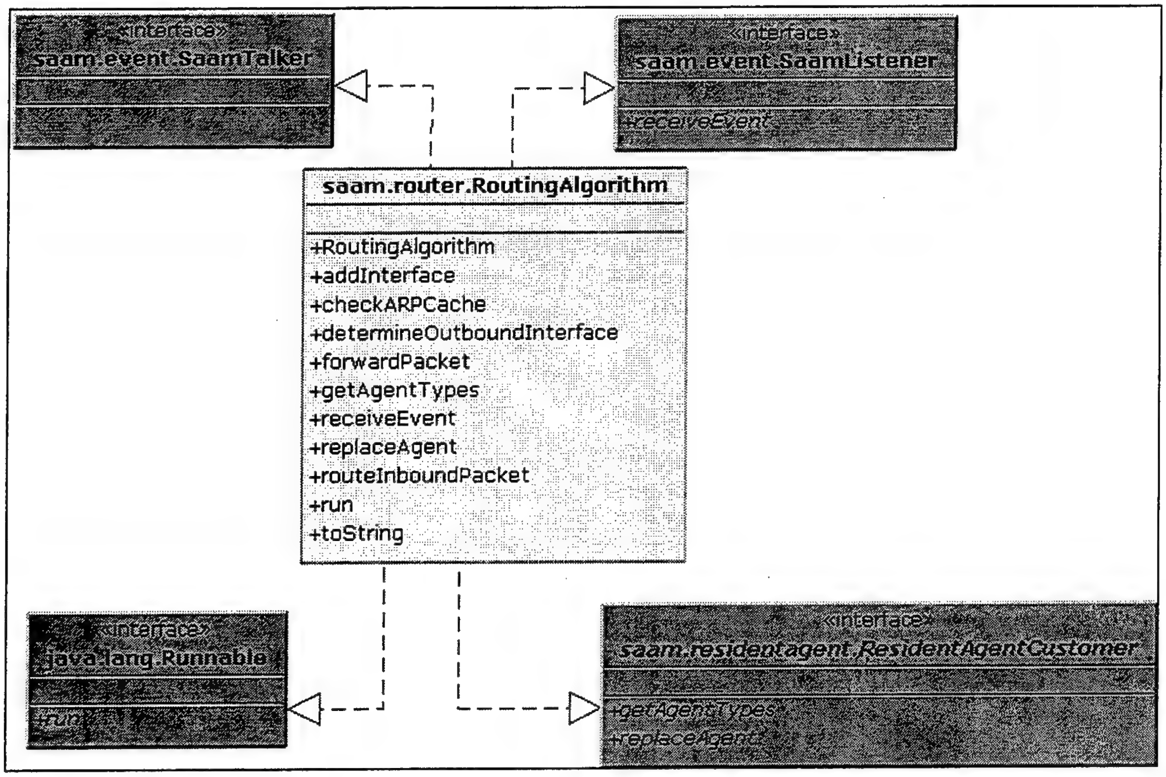


Figure 35. saam.router.RoutingAlgorithm

(1) java.lang.Runnable. The Runnable interface is defined in the standard Java Developer's Kit (JDK) in the package java.lang. This interface requires its implementors to provide a run method, which allows them to operate on a separate thread.

(2) saam.residentagent.ResidentAgentCustomer. The ResidentAgentCustomer interface is intended for use by objects that need access to resident agents on the router. This interface provides a mechanism by which an implementer can be notified when a resident agent the implementor needs access to is replaced. The ResidentAgentCustomer interface provides two methods that must be implemented: getAgentTypes and replaceAgent.

ResidentAgentCustomers must register with the ControlExecutive in order to be notified as to when a particular resident agent is replaced. Calling the registerCustomer method of the ControlExecutive does this. The getAgentTypes method should provide an array of String names of the types of resident agents the customer is interested in. The ControlExecutive will call this method during the registration process

to build a table of agents and their associated customers. When an agent listed in this table arrives, the ControlExecutive calls the replaceAgent method for each customer of that agent, passing it an instance of the new agent.

(3) Inbound Traffic.

(a) Channels

As router Interfaces are instantiated, the RoutingAlgorithm registers to listen on the Channel designated as the Channel that passes notifications from those Interfaces to the RoutingAlgorithm. Adding the number of Interfaces that have been previously instantiated to the ENQUEUING_INBOUND_PACKET_START_CHANNEL defined in the ProtocolStackEvent class determines the appropriate Channel. For example, when a new Interface is instantiated, if the ENQUEUING_INBOUND_PACKET_START_CHANNEL is defined as 90000 and this is the fourth Interface to be instantiated, the RoutingAlgorithm would register to listen on Channel 90003 (90000 + three previous Interface instances).

To forward the packet to the TransportInterface, the RoutingAlgorithm registers to talk on the APPLICATION_ CHANNEL defined in the ProtocolStackEvent class.

(b) Packet Processing

The RoutingAlgorithm is designed to continually sweep packets from the inbound queues of the Interfaces on the router. If the routing algorithm finds that all inbound queues are empty, it waits for a notification from an Interface that a packet is about to be queued, at which time the RoutingAlgorithm wakes up and repeat the cycle.

When the RoutingAlgorithm dequeues an inbound packet, it first compares the packet's IPv6 destination to the IPv6 addresses of the local Interfaces, if the packet is destined for one of these Interfaces, the packet is forwarded to the TransportInterface for processing. Otherwise, the packet is routed to the next hop as described below.

(4) Outbound Traffic.

(a) Channels

In addition to forwarding packets not destined for the router on to the next hop, the Routing Algorithm is also capable of receiving outbound packets from the TransportInterface. To receive packets from the TransportInterface, the RoutingAlgorithm registers to listen on the FROM_TRANSPORTINTERFACE_TO_ROUTINGALGORITHM_ CHANNEL in the ProtocolStackEvent class.

When a router Interface is instantiated, the RoutingAlgorithm registers to talk on the Channel that is designated as the Channel that passes notifications from the RoutingAlgorithm to that Interface. Adding the number of Interfaces that have been previously instantiated to the FROM_ROUTINGALGORITHM_TO_INTERFACE_START_ CHANNEL defined in the ProtocolStackEvent class determines the appropriate Channel.

(b) Packet Processing

To route outbound packets, the RoutingAlgorithm consults the FlowRoutingTable and ARPCache to determine the information necessary to route the packet to the next hop. The RoutingAlgorithm also determines which local Interface to forward the packet to by comparing the network portion of the next hop to the network portions of the IPv6 addresses of each local interface. The packet is then forwarded to the outbound Interface for queuing.

e) The Scheduler

There is one Scheduler (Figure 36) for every Interface on a router which performs the link layer functions and scheduling for outbound packets. The Scheduler only schedules outbound traffic. It contains the logic that determines how to extract packets from the service level queues assigned to the router Interface associated with that Scheduler. Like the RoutingAlgorithm, the Scheduler implements the SaamTalker, SaamListener, and Runnable interfaces.

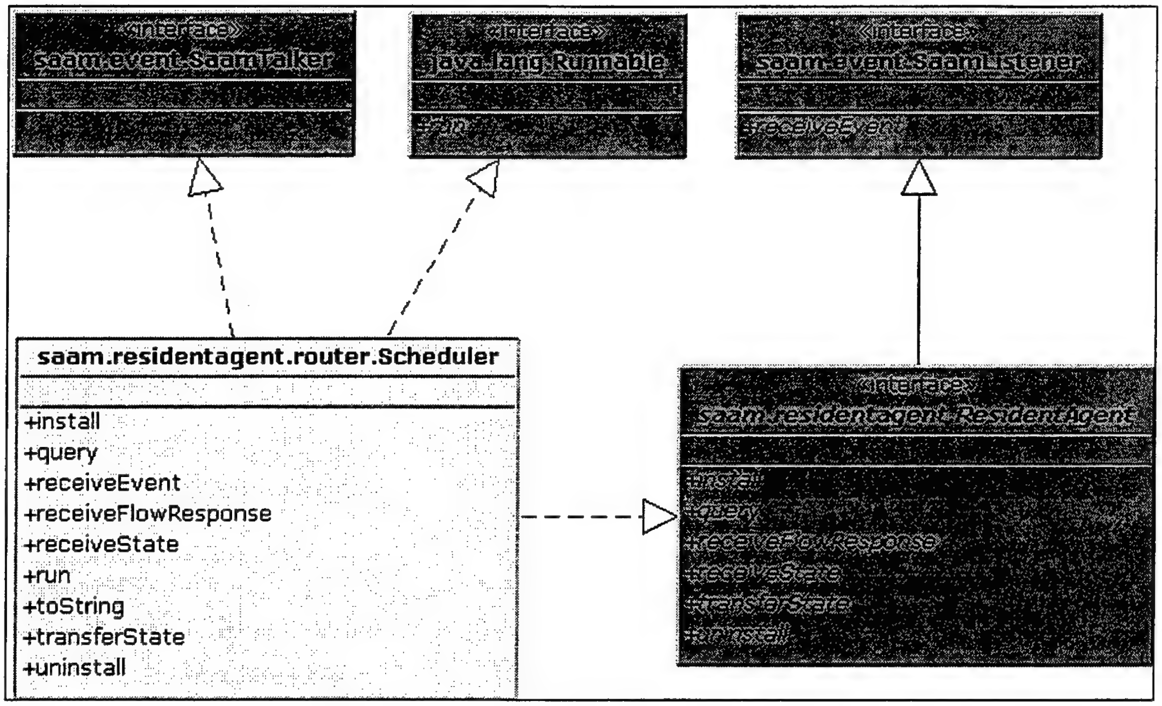


Figure 36. saam.residentagent.router.Scheduler

The Scheduler also implements the ResidentAgent interface. This is what enables dynamic replacement of the scheduling algorithm. At any time, the server or the demo station can send a saam.residentagent.router.Scheduler object to replace the all instances of the existing agent with that name on a router.

(1) Channels. When this resident agent is installed, the Scheduler registers to listen on the Channel that has been designated to pass traffic to the Scheduler from the router Interface assigned to it. Calling the getFromInterfaceToSLQueueChannel method defined in the ProtocolStackEvent class determines the appropriate Channel.

To forward the packet to the NetworkInterfaceCard, the Scheduler registers to talk on the appropriate Channel by calling the getFromInterfaceToSLQueueChannel method in the ProtocolStackEvent class and passing the instance number of the Scheduler as the only parameter.

The Scheduler also registers to talk on a Channel designated for notifying any interested listeners that a packet has been dequeued from a service level

queue. There is a series of Channels devoted to this type of traffic. The first of these is the FROM_SLQUEUE_ TO_SCHEDULER_START_CHANNEL. By adding the instance number of the Scheduler to this number, an interested listener can determine which Channel that Scheduler is sending notifications on.

(2) Packet Processing. The algorithm used by the Scheduler to schedule packets is entirely up to the designer of the resident agent. We have provided a simple algorithm that dequeues packets on a round-robin basis, emptying the highest priority queue first and then proceeding on to the next-highest priority queue. The logic for this algorithm resides in the run method of the Scheduler.

f) The TransportInterface

There is one TransportInterface (Figure 37) for every router which performs the transport layer functions for that router. The TransportInterface serves as the deliver mechanism for applications listening to the router's *emulated* UDP ports. The TransportInterface class implements both the SaamTalker and SaamListener interfaces.

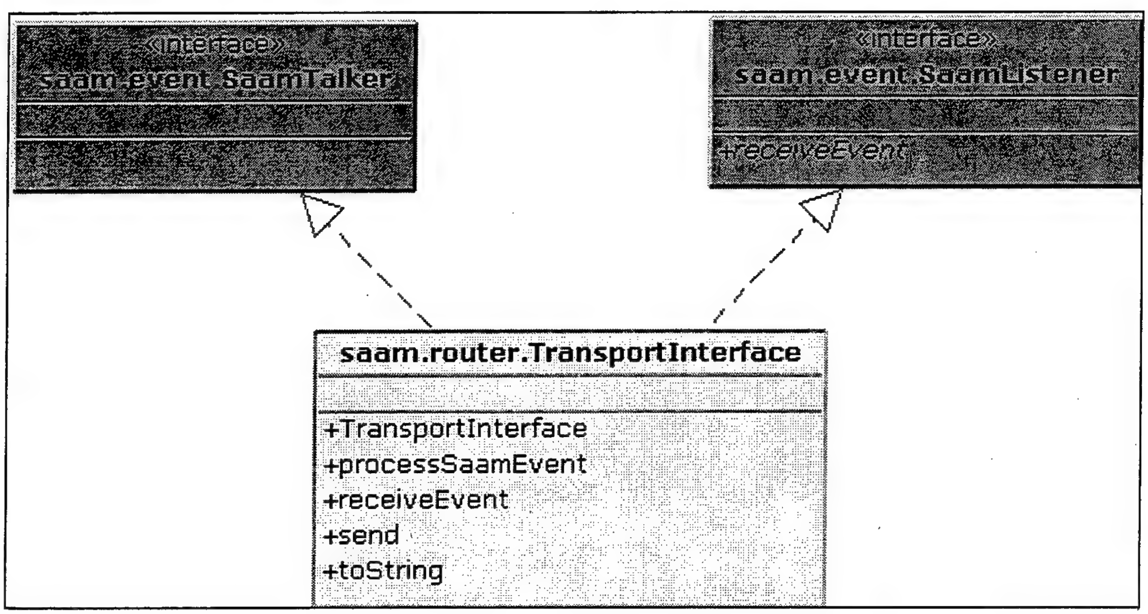


Figure 37. saam.router.TransportInterface

(1) Inbound Traffic.

(a) Channels

To receive inbound packets from the RoutingAlgorithm, the TransportInterface registers to listen on the FROM_ROUTINGALGORITHM_TO _TRANSPORTINTERFACE_CHANNEL defined in the ProtocolStackEvent class.

Upon instantiation, the TransportInterface registers to talk on the PACKETFACTORY_CHANNEL defined in the ProtocolStackEvent class. This is the Channel designated for traffic destined for the PacketFactory.

For packet delivery on an *emulated* port, the TransportInterface is registered by the ControlExecutive to talk on a port whenever an application registers to listen on that port. To do this, the TransportInterface is simply registered to listen to the Channel with the ID that corresponds to the port number.

(b) Packet Processing

The TransportInterface strips the UDP Header of inbound packets and, if there is an application listening on the *emulated* port, forwards the packets on the *emulated* port for which the packet was destined.

When the TransportInterface receives an inbound packet that is destined for the SAAM_CONTROL_PORT defined in the ControlExecutive, the packet is delivered to the PacketFactory for processing.

(2) Outbound Traffic.

(a) Channels

To forward outbound packets to the RoutingAlgorithm, the TransportInterface registers to talk on the FROM_ TRANSPORTINTERFACE _TO_ ROUTINGALGORITHM _CHANNEL defined in the ProtocolStackEvent class.

(b) Packet Processing

For outbound packets, the TransportInterface appends the UDP Header to the packet and then forwards the packet to the RoutingAlgorithm.

5. The PacketFactory

The PacketFactory (Figure 38) is the object that is used to construct SAAM packets for sending, and to decompose when they are received.

a) Constructing SAAM Packets

The PacketFactory provides a public no-args constructor and three public methods that can be used by applications to build SAAM packets. The first of these methods accepts an instance of a Message, extracts the state of the Message in the form of a byte array, and appends that byte array to the byte array that represents the packet being built. The second append method accepts a string that represents the fully qualified name of a classfile that implements the ResidentAgent interface.[8] The PacketFactory uses this string to determine the location of the class file (<filename>.class) on the local machine. It then reads in the byte array representation of the classfile and appends it to the packet being built. The third append method accepts an instance of a class that implements the ResidentAgent interface, determines the fully qualified name of the agent, and then calls the second append method to append the agent.

[8] "saam.residentagent.router.Scheduler" is an example of a fully qualified classfile name that is passed to the PacketFactory in order to send the bytecode representing the ResidentAgent.

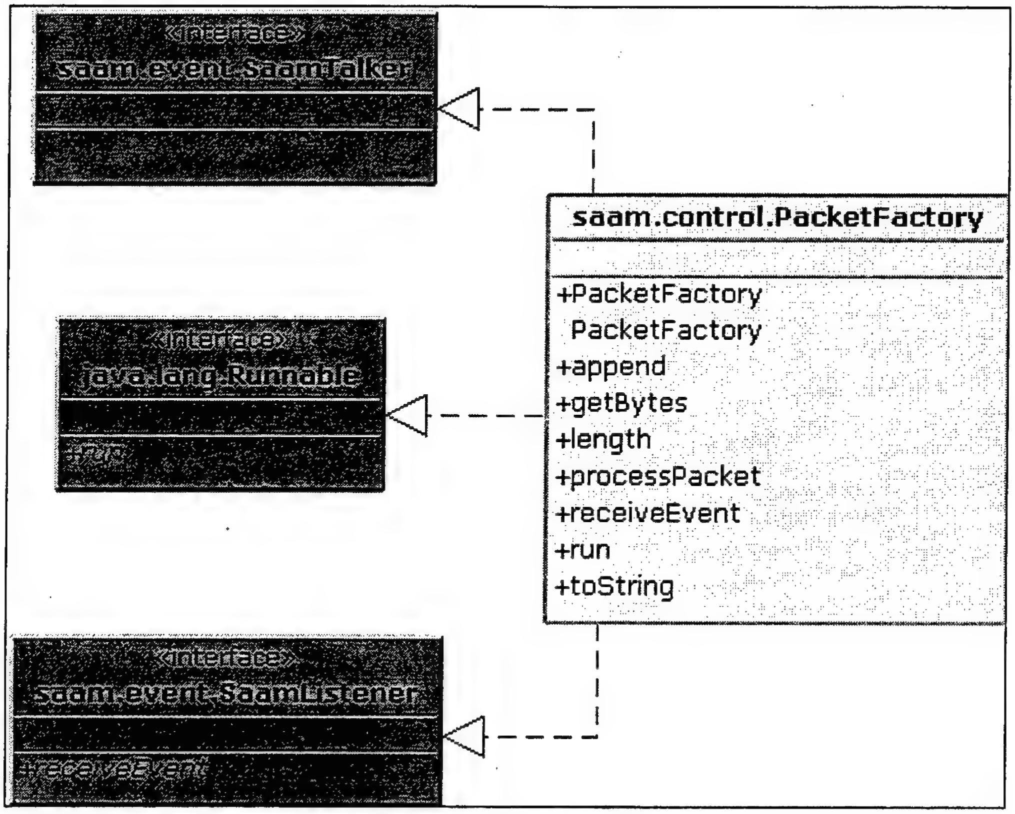


Figure 38. saam.control.PacketFactory

To extract the byte array representation of the SAAM packet from the PacketFactory, the getBytes method must be called. This method first appends a SAAM header to the packet (Figure 13), then it sets a boolean flag[9] indicating that the packet has been retrieved, and finally, it returns the byte array representation of the SAAM packet.

b) Receiving SAAM Packets

The PacketFactory also provides a constructor that is visible only to objects in the saam.control package which is used by the ControlExecutive to receive SAAM packets and decompose them into their individual messages and resident agents. When instantiated by the ControlExecutive, the PacketFactory registers to monitor the PACKETFACTORY_CHANNEL in the ProtocolStackEvent class. The Translator uses this Channel to forward SAAM packets that arrive on the emulation port; and the

[9] This boolean flag is used by the append methods to reinitialize the packet if the getBytes method has been called. This allows multiple SAAM packets to be created with the same instance of the PacketFactory.

TransportInterface uses it to forward SAAM packets that arrive from the emulated protocol stack.

The PacketFactory receives inbound packets from its receiveEvent method. When a packet arrives, the thread used to process inbound packets wakes up and calls a private method to process the packet. This private method (processPacket) then determines the number of updates in the arriving packet and proceeds to process each update individually.

Message objects are instantiated by the PacketFactory directly[10] and then forwarded to the ControlExecutive on the SAAM_CONTROL_PORT which can be found in the ControlExecutive. Resident agents are not directly instantiated by the PacketFactory. The PacketFactory merely defines the class and passes the class definition to the ControlExecutive. This allows the ControlExecutive to first examine the agent to determine whether or not it violates any policy issues.

---

[10] The process of instantiating Messages should eventually be moved into the ControlExecutive where Messages could first be analyzed to determine whether or not they violate any policy issues.

VI. INTEGRATION

In order to enable nodes within a SAAM network to communicate with one another, it is important to understand how the components of the architecture fit together. The emulated protocol stack is the delivery mechanism for SAAM traffic, while resident agents provide the bond that ties applications utilizing this protocol stack together.

A. PLACING AN APPLICATION ON TOP OF THE PROTOCOL STACK

Applications can utilize the emulated protocol stack in two ways. If the application includes multiple class files, a resident agent should be created to instantiate the application. If the application consists of one classfile and is small enough that it could easily be delivered over the network, the application could implement the ResidentAgent interface and be passed to the protocol stack.

This section first describes what is required to create a resident agent, then it provides examples of the two uses of these agents described above.

1. Creating a Resident Agent

Implementing the ResidentAgent interface (Figure 39) is all that is necessary to create a resident agent that can operate within this prototype. The ResidentAgent interface prescribes the overriding of six methods to its implementors.

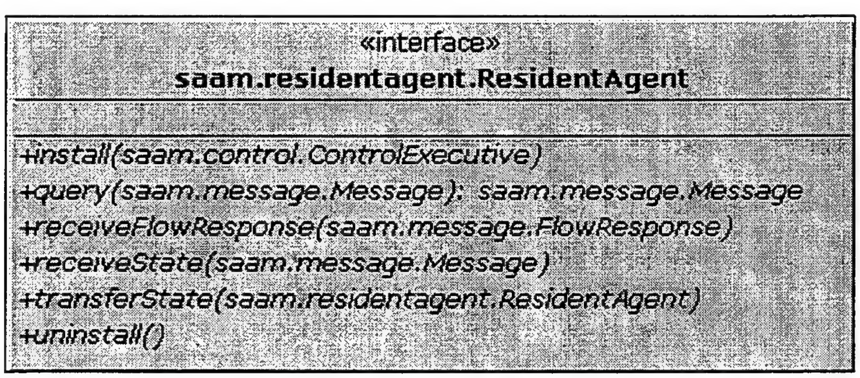


Figure 39. The ResidentAgent Interface

a) *Install*

The install method receives a ControlExecutive instance as its only parameter. When this method is called, the agent can use the ControlExecutive passed to it to performs such tasks as registering itself to process messages, registering to listen on protocol stack channels, registering to monitor emulated ports, or sending traffic across the network.

The following code sample provides a simple implementation of the install method. First, it registers with the ControlExecutive as a MessageProcessor. When the call is made to `registerMessageProcessor`, the ControlExecutive will first call the `getMessageTypes` method within the agent to determine the types of messages this agent is interested in processsing, then it will determine if this agent is allowed to process those types of messages, and finally, if the agent is authorized, the ControlExecutive will add the agent and its message types to the registration table.

```
            public void install(ControlExecutive
controlExec){
                  this.controlExec=controlExec;

                   controlExec.registerMessageProcessor(this);

                   //this channel is used by the first instance
of a router Interface
                   //to send notifications when it is about to
enque an inbound packet
                   int channel = ProtocolStackEvent.
                        ENQUEUING_INBOUND_PACKET_START_CHANNEL;

                   //here we register to monitor the channel we
just assigned above.

controlExec.addListenerToChannel(this,channel);
            }
```

Once this agent has registered as a MessageProcessor, it identifies the integer that represents the Channel the agent seeks to monitor by calling a static final int in the `ProtocolStackEvent` class. Once the integer is retrieved, the agent passes it as a parameter to the `addListenerToChannel` method in the ControlExecutive. The ControlExecutive will again screen the agent to determine if it is authorized to monitor the Channel and then place the agent in the Channel's Vector of listeners.

b) Query

Some resident agents are accessed by Objects on the router. This method provides the means for communication between an Object on the router and this ResidentAgent. The implementations of this method can vary greatly. All that is required is that the agent receives a Message as its only parameter and that it returns a Message. The following sample code provides a simple implementation. The comments provide the narrative.

```
//this method allows an object to perform a table
//lookup, retrieving
//a message that represents an entry in the table
public Message query(Message message){

    //for demo purposes only, we assume the
  incoming Message is a
  //FlowRoutingTableEntry

    int flowID =
((FlowRoutingTableEntry)message).getFlowID();

    //Here, the agent extends java.util.Hashtable,
so it inherits the
    //the get method.  This call to get returns
an Object that is cast
    //to a FlowRoutingTableEntry.
    FlowRoutingTableEntry result =
(FlowRoutingTableEntry)
      get(new Integer(flowID));

    return result;
  }
```

c) ReceiveFlowResponse

When a ResidentAgent requests a flow by calling the `requestFlow` method of the ControlExecutive, the agent expects to be assigned a flow and to be notified when that flow is assigned. This method provides a mechanism for notifying the ResidentAgent that a FlowResponse has arrived.

d) ReceiveState

The `receiveState` method is intended to operate in conjunction with the `transferState` method. This method is called one or more times from within the

`transferState` method of an agent that is about to be replaced by this agent. The purpose of this method is to enable a state transfer from the old agent to the new agent. This method receives a Message as its only parameter. One of these messages could contain all information that is intended to be transferred, or it may only contain partial information (a row in a table, for example).

e) *TransferState*

When an agent is about to be replaced, the ControlExecutive calls the `transferState` method, passing the old agent a copy of the new agent. The old agent should then call the receiveState method on the new agent, and pass to the new agent any messages that should be passed.

f) *Uninstall*

When an agent is being replaced, the ControlExecutive will remove the old agent from all channels it is registered to talk on and from all channels or emulated ports it is monitoring. The `uninstall` method is called by the ControlExecutive upon replacement in order to allow the old agent a chance to perform any other cleanup that might be necessary, such as disposing of a graphical user interface, for instance.

2. Creating a Resident Agent to Call an Application

An object that implements the ResidentAgent interface can instantiate other objects that exist on the host that agent object is destined for. This is the implementation we used to instantiate the Server prototype. The simple agent we created instantiated the Server class, registered with the ControlExecutive to process three Message types: Hello, FlowResponse, and LinkStateAdvertisement. Once registered, this agent would receive those messages and pass them up to the Server by invoking the appropriate method on the Server. This resident agent is provided in Appendix G. The fully qualified class name is `saam.residentagent.server.ServerAgent`.

3. Creating a Resident Agent to Act as an Application

In Appendix G, we have provided several examples of resident agents that perform various tasks such as: Acting as routing tables, acting as threaded packet schedulers, acting as applications that generate network traffic, and acting as remotely deployed packet sniffers.

B. UTILIZING THE PROTOCOL STACK

The Control Executive is the base component (similar to the kernel of an operating system) of a router. As such, it will make sure only resident agents from a valid SAAM server are installed and allowed to communicate using the emulated protocol stack. The Control Executive controls communication over the protocol stack by providing the only public methods for sending and receiving SAAM network traffic in the whole SAAM package.

1. Passing the Control Executive to An Application

In order to send or receive traffic over the prototyped SAAM network, an application must be provided with an instance of the Control Executive. This can be done in two ways: If an application is a resident agent, the application will receive an instance of the Control Executive when it is installed on the host it is destined for; or, if an application is instantiated by a resident agent, the instantiating agent can pass the application an instance of the Control Executive.

2. Sending Traffic

The Control Executive contains three methods that allow applications to send traffic over the network. These methods allow applications to send IPv6 packets, SAAM packets, or subclasses of the Message class.

a) *Sending IPv6 Packets*

Objects that have been assigned flows can send IPv6 packets with this method:

`public void send(java.lang.Object sender, IPv6Packet ipv6Packet).`

In order to use this method, Objects must first request a flow using the `requestFlow` method; and then receive a flow assignment from the server. The `sender` parameter represents the Object sending the message, while the `ipv6Packet` parameter represents the `IPv6Packet` to be sent on the flow contained in the header of the IPv6 packet being sent.

b) *Sending Messages*

Objects that have been assigned flows can send Messages with this method:

`public void send(java.lang.Object sender, Message message, int flowID, short sourcePort, IPv6Address destHost, short destPort).`

In order to use this method, Objects must first request a flow using the `requestFlow` method; and then receive a flow assignment from the server. The `sender` parameter represents the Object sending the message. The `message` represents the subclass of Message to be sent. The `flowID` parameter represents the flow ID that has been assigned for traffic from sender destined for `destHost`. The sourcePort is the port on the local machine that sender is listening on. The `destHost` is The IPv6 address of the destination. The `destPort` is the port to which `destHost` is listening.

c) *Sending SAAM Packets*

Objects that have been assigned flows can send SAAMPackets with this method:

`public void send(java.lang.Object sender, SAAMPacket saamPacket, int flowID, short sourcePort, IPv6Address destHost, short destPort).`

The only difference between this method and the previous one is that this method accepts a SAAM packet as a parameter vice a subclass of Message. This is for applications that desire to send multiple combinations of messages and resident agents in one packet.

3. Receiving Traffic

The Control Executive contains two methods that allow applications to receive traffic over the network and one method allowing applications to register as listeners to channels that are not ports. These methods allow applications to send IPv6 packets, SAAM

a) Monitoring Ports

Objects implementing the SaamListener interface can use the following method to receive traffic from a host across the network:

```
public void monitorPort(SaamListener listener, int port).
```

In order to send a flow request or any other type of traffic in a SAAM network, sending Objects must be listening to a port. The following method assigns Objects a random port that is higher than the highest well-known port, but no higher than MAX_PORT:

```
public int listenToRandomPort(SaamListener listener)
```

b) Monitoring Channels

SaamListeners use the `addListenerToChannel` method to attach themselves as a listener to a Channel. If this method succeeds, the listener will receive all events that are sent on this Channel.

VII. RESULTS

This results chapter describes the test environment that was used to test the integration of the emulated routers and server. The first section describes the topology of the test bed. The second section highlights the type of performance numbers that were recorded during the operation of the server.

A. TEST BED

We developed a test bed in which to test the basic functionality outlined in Chapter V. There are five participants in the test: A demo station, an emulated router with a server on it, and three emulated standalone routers. Using these components, we were able to demonstrate the following functionality: Control traffic passing between the routers and the server; Resident agent installation, activation, de-activation, uninstallation, and replacement; Flow assignments; and Flow usage by applications residing on different hosts. Figure 40 illustrates the components and the connections between them.

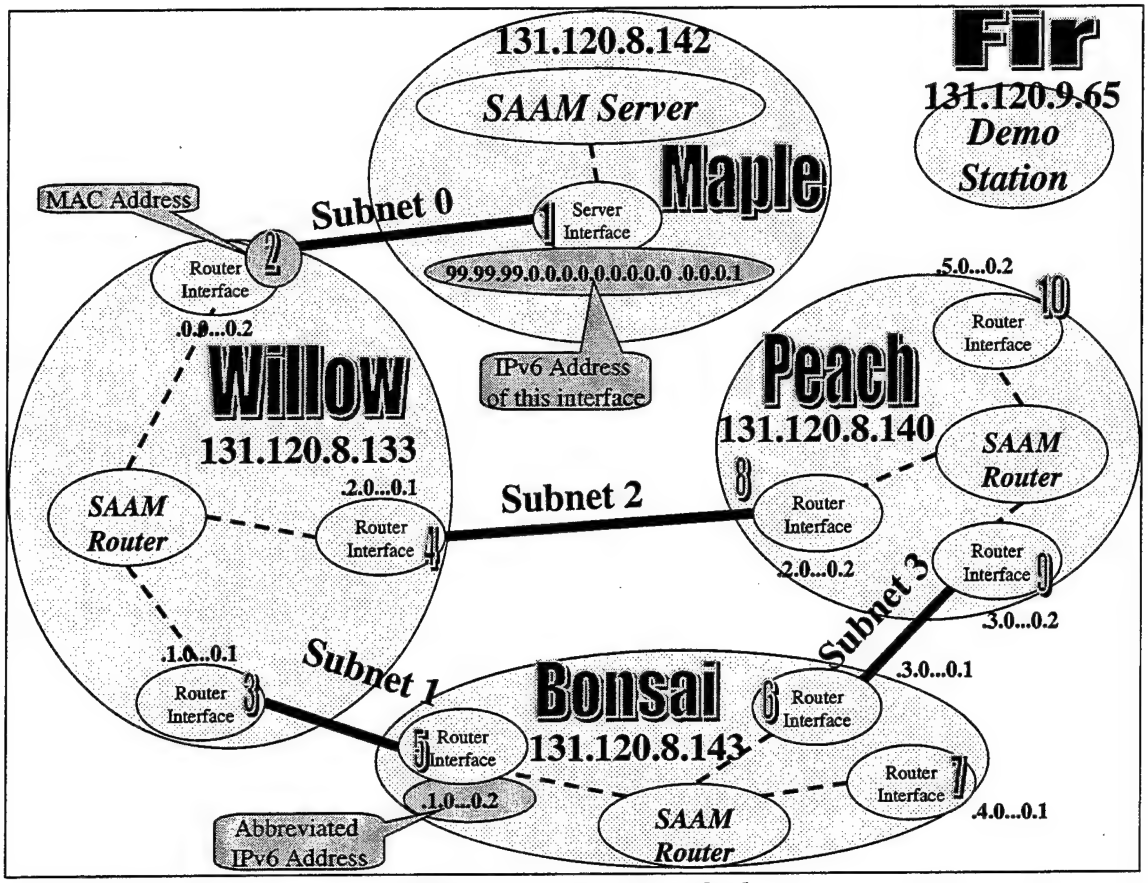


Figure 40. SAAM Testbed

1. Demo Station

The demo station is used to initialize the nodes in the test SAAM testbed. Translators are first instantiated on the machines that host the server and the routers. The Translators initially listens on the emulation port (9002) for initialization data from the demo station. Once this data is received and all core routing components were instantiated, the ControlExecutive at each router is informed by the Translator that the router was ready. After this initialization, the Translator began listening on the SAAM port (9001) for SAAM traffic.

The demo station is located on "Fir" whose IPv4 address 131.120.9.65. The demo station is hard-coded with all of the information necessary to initialize the server and all routers. Building and transmitting SAAM packets at the demo station proved to be a simple task. All that is required in the demo station class file was the appropriate

topology configuration, a network socket connection, and an instance of the `PacketFactory`.

This topology configuration information is used to instantiate messages which are then appended to the outbound packet. Resident agents were appended to the packet using the append method that takes a string as its only parameter.[11]

2. Server

For this test bed, we decided to instantiate the server as a resident agent on a router. The router contains one interface. Otherwise, a separate, stripped-down protocol stack would be needed. As described in Chapter IV, the server should be able to execute on a router, so this testbed situation enabled us to more fully exercise this functionality. Our server resides on "Maple" whose IPv4 address is 131.120.8.142. The `Interface` it received from the demo station has an IPv6 address of 99.99.99.0.0.0.0.0.0.0.0.0.0.0.0.1 and MAC address[12] of 1.

a) Emulation Table Entries

Figure 41 shows the contents of the server's emulation table upon initialization from the demo station. Since the server has only one interface, it only needs to know the routing information about the single interface that it is directly connected to on "Willow."

---

[11] When appending resident agents, it is necessary to ensure that the classfile representing the resident agent is located in the proper place on the file system. For example, the classfile for the agent"saam.residentagent.router.Scheduler" should be saved as saam\residentagent\router\Scheduler.class on a Windows platform.

[12] The current architecture only supports a one-byte MAC address. This is very limiting and should be expanded to more accurately reflect reality.

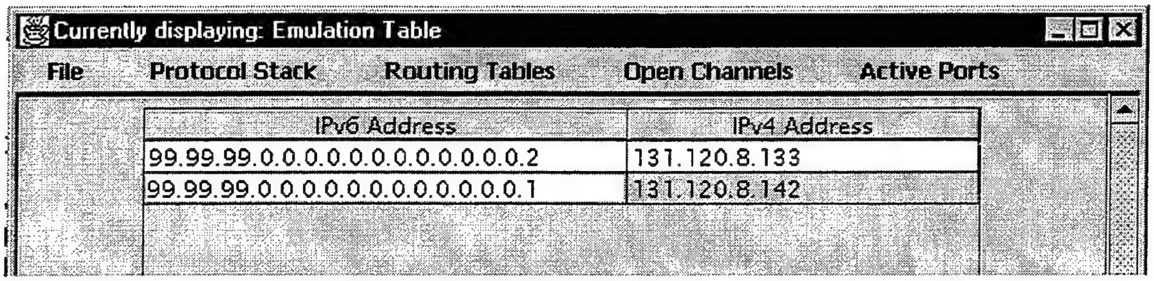


Figure 41. The Server's Emulation Table

Notice, however, that the emulation table for the server has two entries. This is a side-effect of placing the server on top of an emulated router. Since every router sends a Hello Message when it is ready to route, the router upon which the server is built must be equipped to send the Hello to itself. We found it very convenient to be able to alter the routing tables, thereby providing the same effect as altering the code within the router itself. This provided an excellent example of the portability of the router code, since it allows the demo station to easily relocate the server by merely sending the table updates to a different host.[13]

b) ARP Cache Entries

The server's ARP cache (Figure 42) contains two entries as well. One entry is used to determine the MAC address of the interface on "Willow" that the server's interface is directly connected to, while the other is used as a loopback entry.

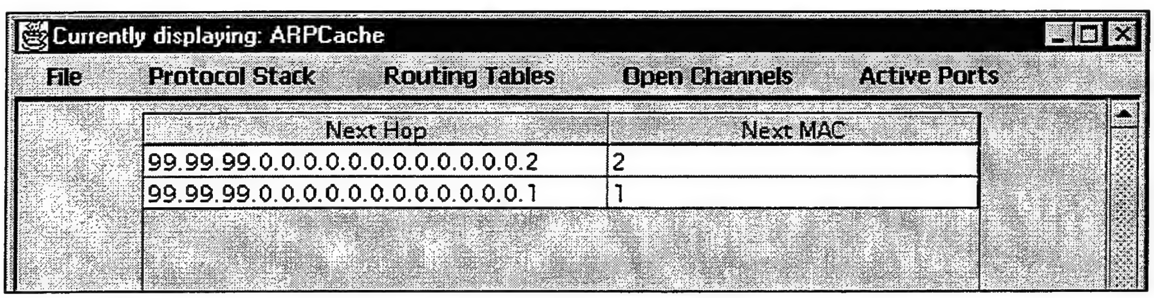


Figure 42. The Server's ARP Cache

[13] The routing code was improved after this writing to enable the `ControlExecutive` to recognize when an outbound packet is destined for a local interface and to forward such a packet directly to the `TransportInterface` where it was then treated as an inbound packet and forwarded on the appropriate destination port.

c) *Flow Routing Table Entries*

The server's flow routing table (see Figure 43) initially contains only the entry that is used to provide the service level and next hop associated with flow zero.[14]

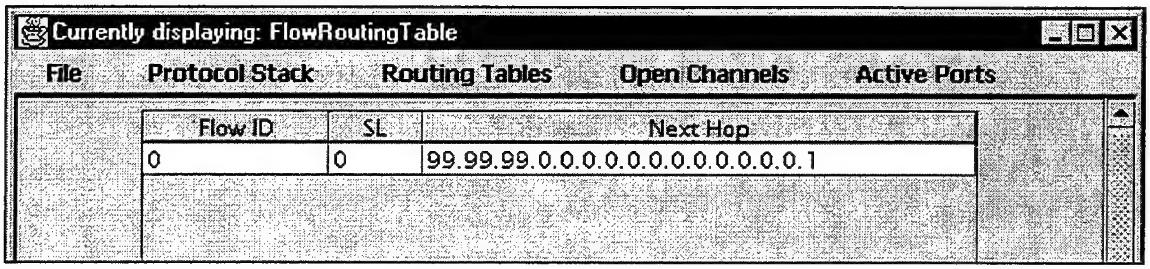


Figure 43. The Server's Flow Routing Table

3. Willow's Router

The first emulated router is instantiated on "Willow" with IPv4 address 131.120.8.133. This router is given three interfaces with the IPv6 and MAC addresses shown in Table 6. From Figure 41, it is evident that this router is directly attached to every other node in the topology. This router serves as the server's default gateway and provides a path between the server and the other two routers.

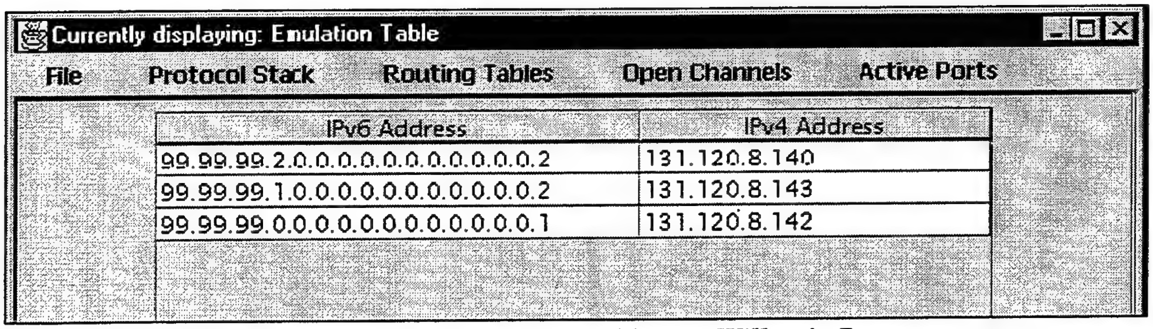
IPv6 Address	MAC
99.99.99.0.0.0.0.0.0.0.0.0.0.0.0.2	2
99.99.99.1.0.0.0.0.0.0.0.0.0.0.0.1	3
99.99.99.2.0.0.0.0.0.0.0.0.0.0.0.1	4

Table 6. The Interfaces of Willow's Router

[14] Flow zero was mistakenly used in our design as the flow of all control traffic destined for the server. We later discovered RFC 1809, which states, "The zero Flow Label is reserved to say that no Flow Label is being used."

a) *Emulation Table Entries*

The emulation table for this router is initialized by the demo station with the three entries listed in Figure 44. Each of the entries represents a direct connection from one of the local interfaces to an interface on the same subnet as the local interface.

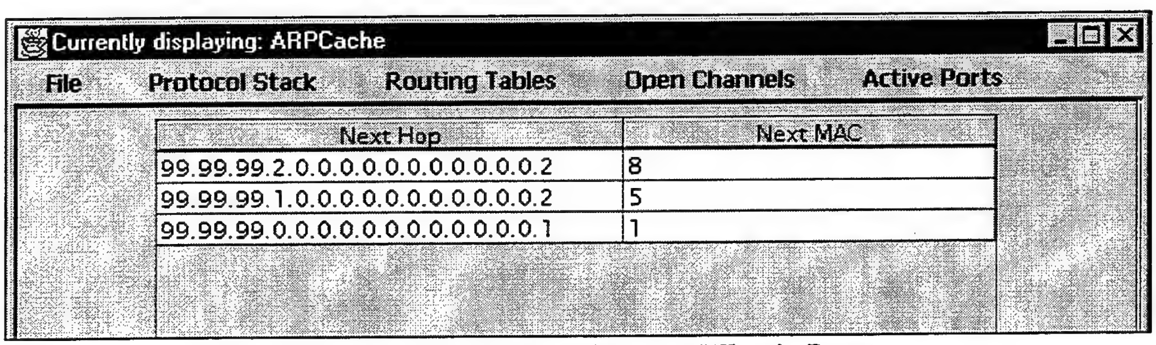


IPv6 Address	IPv4 Address
99.99.99.2.0.0.0.0.0.0.0.0.0.0.0.2	131.120.8.140
99.99.99.1.0.0.0.0.0.0.0.0.0.0.0.2	131.120.8.143
99.99.99.0.0.0.0.0.0.0.0.0.0.0.0.1	131.120.8.142

Figure 44. The Emulation Table For Willow's Router

b) *ARP Cache Entries*

Figure 45 lists the ARP Cache Entries for Willow's router. The first entry contains the next hop and MAC address of the interface it is connected to on Bonsai's router. The second contains information for the interface on Peach's router. The final entry contains the next hop and MAC address of the interface used by the server.



Next Hop	Next MAC
99.99.99.2.0.0.0.0.0.0.0.0.0.0.0.2	8
99.99.99.1.0.0.0.0.0.0.0.0.0.0.0.2	5
99.99.99.0.0.0.0.0.0.0.0.0.0.0.0.1	1

Figure 45. The ARP Cache For Willow's Router

c) *Flow Routing Table Entries*

Since Willow's router is directly connected to the server via subnet zero, all control traffic in this autonomous region will pass through it. Figure 46 shows the initial flow routing table entry for Willow's router.

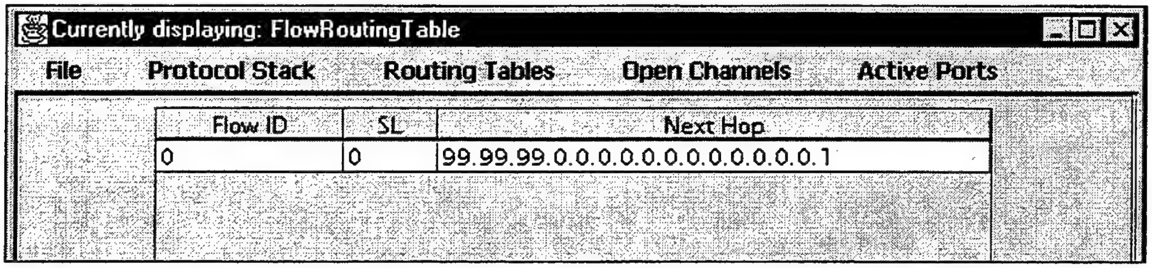


Figure 46. The Flow Routing Table For RouterOne ("Willow")

4. Peach's Router

The second router was placed on "Peach" whose IPv4 address is 131.120.8.140. This router was also given three interfaces as shown in Table 7. From Figure 40, it is evident that this router is directly attached to Willow's router and Bonsai's router. What is not so evident in Figure 40 is that this router acts as a pass-through router for control traffic from Bonsai's router. The flow routing table for router three reveals this subtlety.

IPv6Address	MAC
99.99.99.2.0.0.0.0.0.0.0.0.0.0.0.2	8
99.99.99.3.0.0.0.0.0.0.0.0.0.0.0.2	9
99.99.99.5.0.0.0.0.0.0.0.0.0.0.0.2	10

Table 7. The Interfaces On Peach's Router

a) Emulation Table Entries

The emulation table for this router is initialized by the demo station with only two entries (Figure 47). This router is directly connected to "Bonsai" and "Willow."

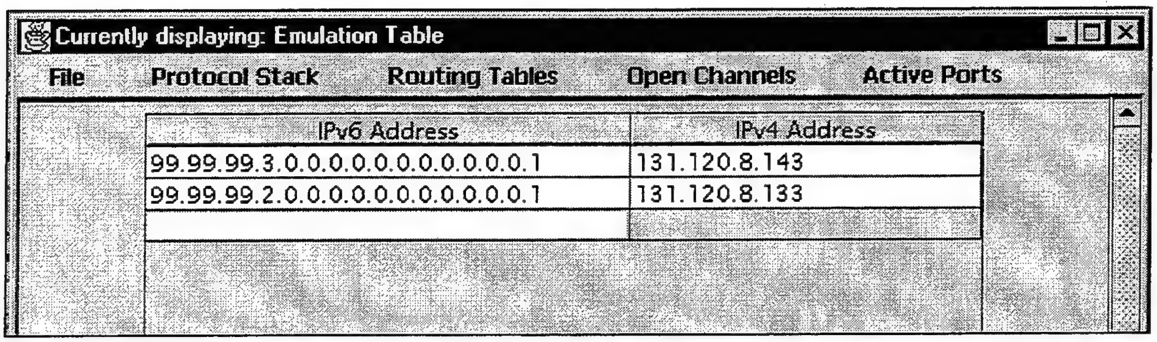


Figure 47. The Emulation Table For Peach's Router

b) ARP Cache Entries

Figure 48 lists the ARP Cache Entries for Peach's router. This ARP cache contains only two entries because this router is only directly connected to two interfaces. The first entry is for the interface on Bonsai's router and the second is for the interface on Willow's router.

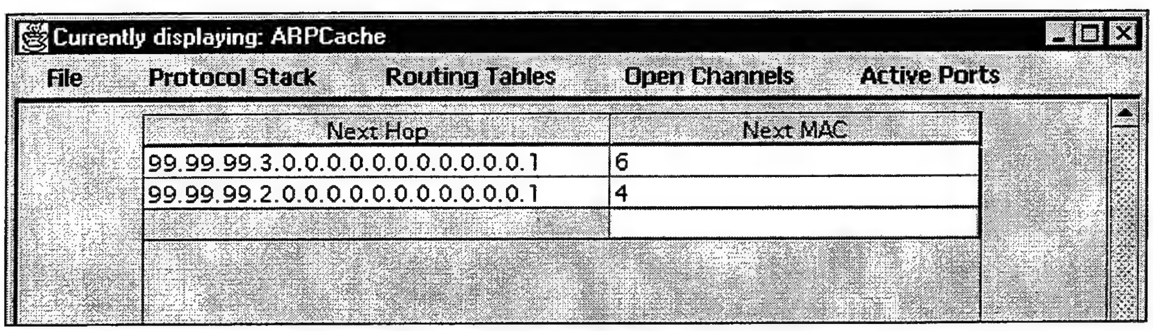


Figure 48. The ARP Cache For Peach's Router

c) Flow Routing Table Entries

The initial flow routing table entry for Peach's router indicates that control traffic initiated by, or sent through this router must pass through Willow's router on its way to the server. Figure 49 shows the initial flow routing table entry for Peach's router.

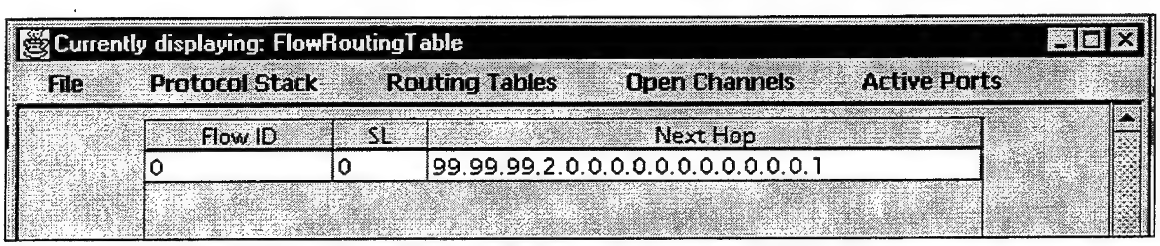


Figure 49. The Flow Routing Table For Peach's Router

5. Bonsai's Router

The third and final router to be instantiated is placed on "Bonsai" with IPv4 address 131.120.8.143. This router is also given three interfaces as shown in Table 8. Figure 40 shows, however, that only two of the interfaces on this router are used in our testbed. The flow routing table for this router shows us that "Peach" is the next hop for control traffic from Bonsai's router.

IPv6Address	MAC
99.99.99.1.0.0.0.0.0.0.0.0.0.0.0.2	5
99.99.99.3.0.0.0.0.0.0.0.0.0.0.0.1	6
99.99.99.4.0.0.0.0.0.0.0.0.0.0.0.1	7

Table 8. The Interfaces On Bonsai's Router

a) *Emulation Table Entries*

Like the emulation table for Peach's router, the emulation table for Bonsai's router is also initialized by the demo station with only two entries (Figure 50). This router is directly connected to "Peach" and "Willow."

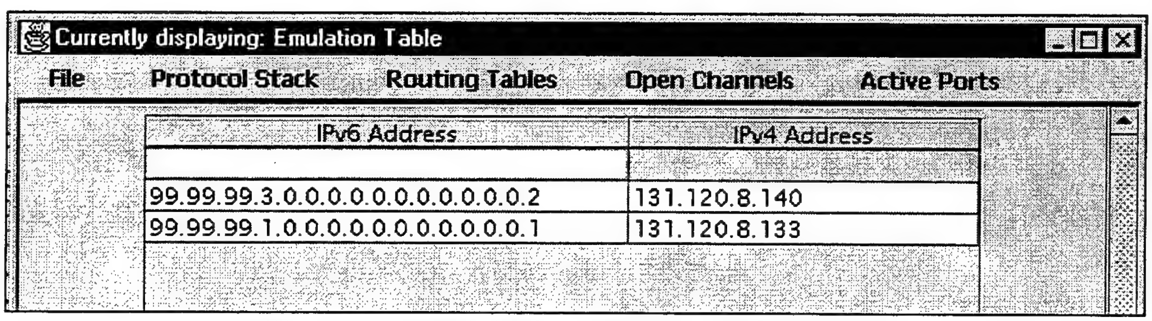
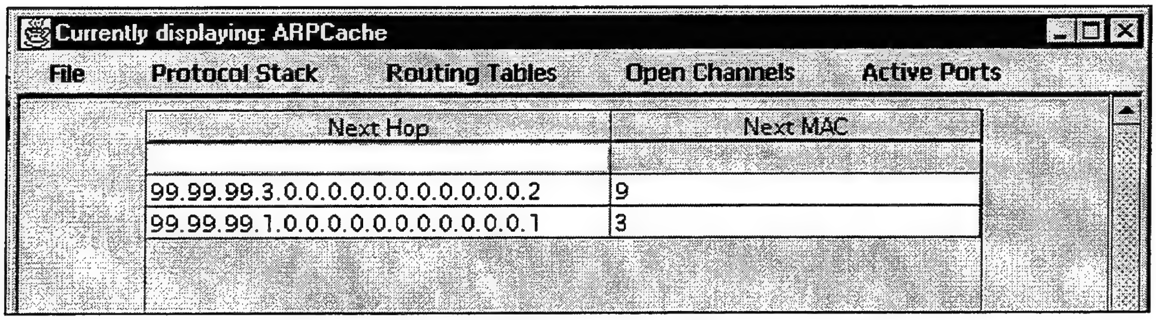


Figure 50. The Emulation Table For Bonsai's Router

b) *ARP Cache Entries*

The ARP Cache (Figure 51) for router three also contains only two entries: One for the interface on Peach's router, and one for the interface on Willow's router.

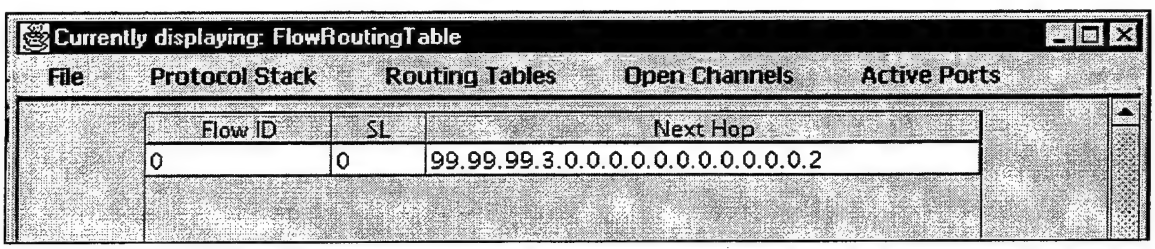


Next Hop	Next MAC
99.99.99.3.0.0.0.0.0.0.0.0.0.0.0.2	9
99.99.99.1.0.0.0.0.0.0.0.0.0.0.0.1	3

Figure 51. The ARP Cache For Bonsai's Router

c) *Flow Routing Table Entries*

The initial flow routing table entry for Bonsai's router indicates that it uses Peach's router to send control traffic to the server. This means that Peach's router must route this traffic to Willow's router, where it will then be routed to the server. Figure 52 shows the initial flow routing table entry for Bonsai's router.



Flow ID	SL	Next Hop
0	0	99.99.99.3.0.0.0.0.0.0.0.0.0.0.0.2

Figure 52. The Flow Routing Table For Bonsai's Router

B. SERVER PERFORMANCE

The performance of the server is considered in phases. The first phase is the initialization phase. This phase is when the physical architecture of the network is being communicated to the server. Whenever routers and/or network segments are added to the network, the server will need to rerun its algorithm for determining all of the possible paths that are now in the network. When routers and/or network segments are removed from the network, the server will simply tag all paths that traverse the removed part of the network as unusable (i.e. no bandwidth, infinite delay and loss rate).

The second phase of server performance is the enhancement of the server's understanding of the network conditions through the use of Link State Advertisements

from the routers. The process of receiving an LSA involves updating service level pipe information itself and then updating the effective QoS of those paths passing over that service level pipe.

The third phase of server performance is the management of user flows. This management of user flows consists of processing of flow requests and the termination of flows. This assignment process includes a relatively quick search through the predetermined paths for one that can meet the QoS requirements. This look-up process combined with the sending of appropriate flow routing table updates and a flow response are the essential functions that must occur. The termination process consists of the removal of the appropriate flow routing table entries as well as the removal of the flow from the PIB.

For this initial iteration, we focused primarily on exercising the activities of the first phase. Four sets of start and finish times were recorded and displayed to gage the performance of the server during this phase. The sequence in which these times were recorded are shown in Figure 53. After the processHello method processes the message, it calls the findAllPossiblePaths method and the determineEffectiveQoSForPaths method. After completing these methods, a flow request from the server to the router that sent the hello message is made.

The second area that we focused on during this iteration was in the assignment of user flows. The time required to respond to these user flow requests was significantly less than that required to establish an initial flow from the server to the newly introduced router described above. This significant difference may be due to the establishment of a new TCP connection to the destination router. The underlying router does not need to establish any new TCP connections to respond to the new user flow request – it simply looks up the appropriate TCP connection from a hashtable.

These number should be viewed as a worst case scenerio due to the developmental environment used. The use of Java when combined with need to emulate an IPv6 router on top of general purpose workstations running Windows NT 4.0 obviously produces a slower system than we can expect from optimized equipment.

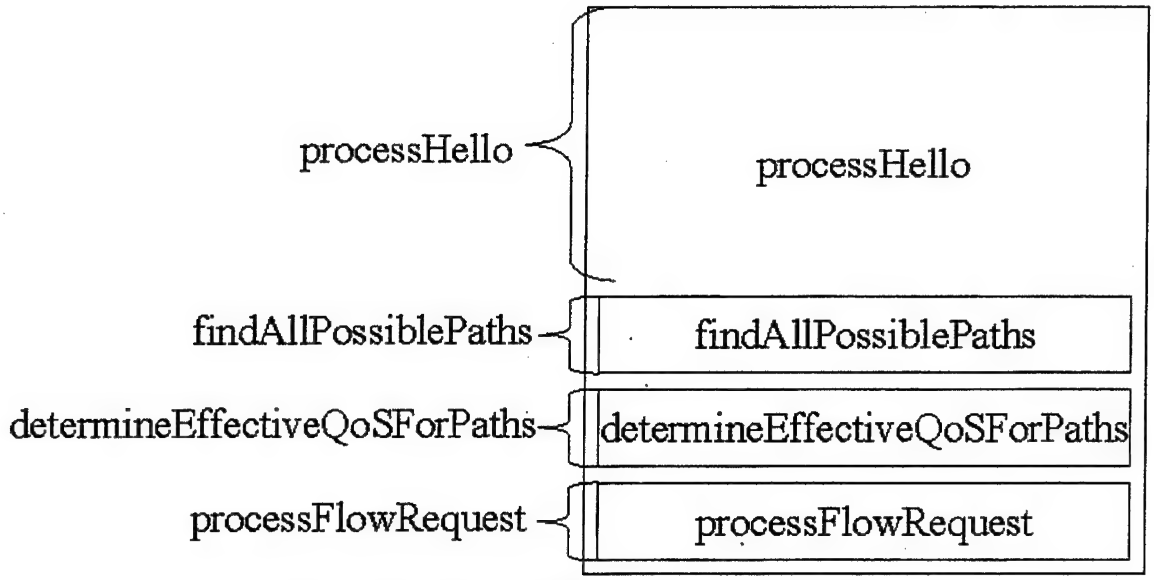


Figure 53. Server Performance Measurements

1. Database Structured PIB

Table 9 and Figure 54 show the performance of the server during the initialization phase when the server is instantiated with a relational database structured PIB.

	findAllPossiblePaths	determineEffectiveQoSForPaths	processFlowRequest	processHello
nodes	Time(ms)	Time(ms)	Time(ms)	Time(ms)
1	140	16	141	282
2	531	422	422	235
3	2047	2531	453	329
4	6000	4594	703	203

Table 9. Performance of the Database Structured PIB During Initialization

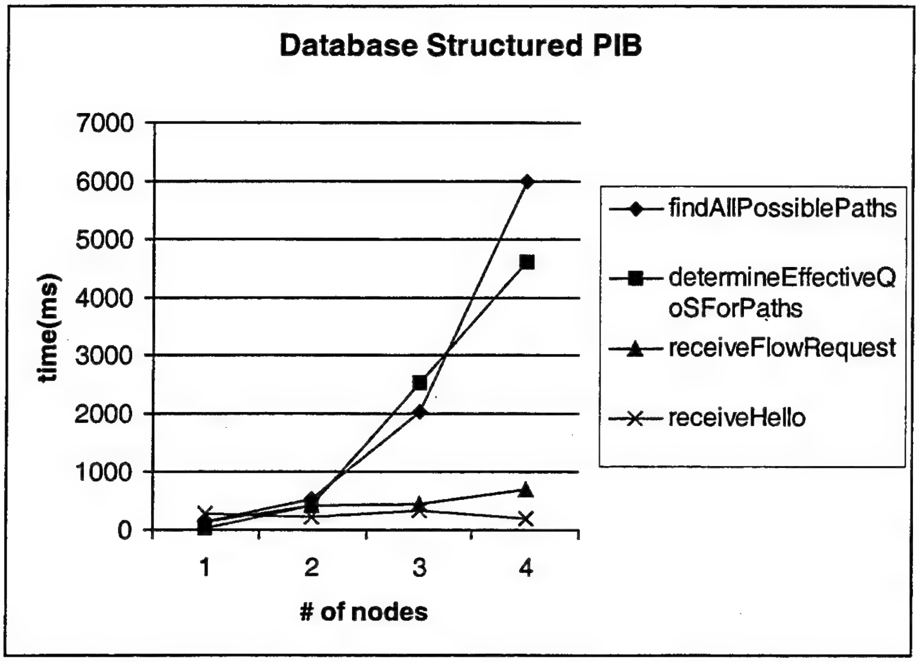


Figure 54. Performance of the Database Structured PIB During Initialization

2. Class Object Structured PIB

Table 10 and Figure 55 shows the average performance of the server during initialization of the test network when the server is instantiated to use a class object structured PIB. These numbers are the average performance numbers over a four trial test.

	findAllPossiblePaths	determineEffectiveQoSForPaths	processFlowRequest	processHello
nodes	Time(ms)	Time(ms)	Time(ms)	Time(ms)
1	46.75	7.75	363.25	43
2	195.25	66.75	343.5	39
3	691.25	105.5	437.5	70.25
4	1410	300.75	417.75	12

Table 10. Performance of the Class Object Structured PIB During Initialization

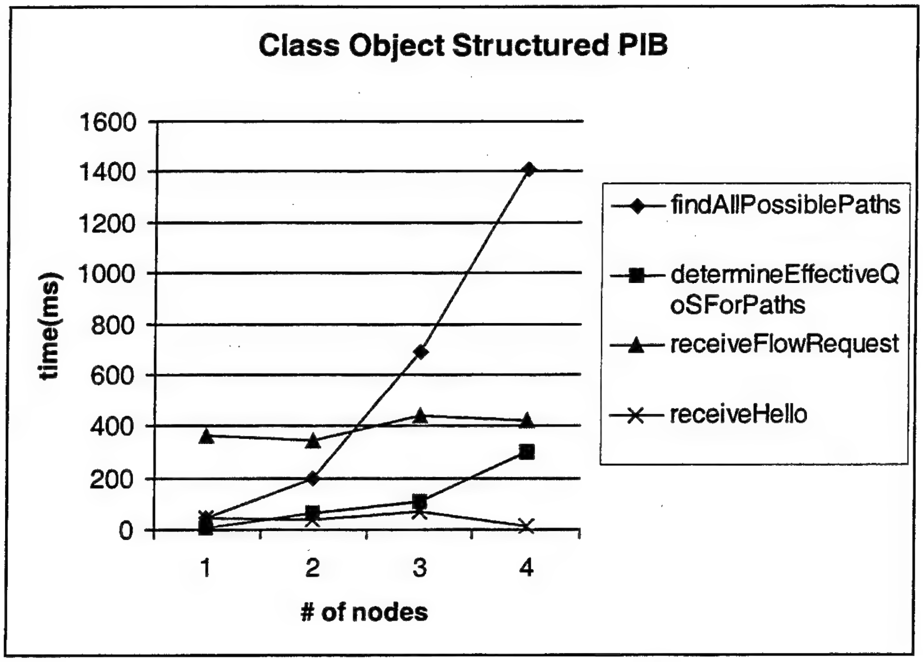


Figure 55. Performance of the Class Object Structured PIB During Initialization

Table 11 and Figure 56 shows the performance of the server during the user flow management phase when it responds to a flow request from the willow host to the cherry host. These number are the actual numbers obtained during the four trials. Notice that the performance during the third trial is a significant outlier that can be safely accredited to other activity on the host computer.

	processFlowRequest
Test #	**Time(ms)**
1	78
2	78
3	203
4	78

Table 11. Performance of the Class Object Structure PIB With User Flows

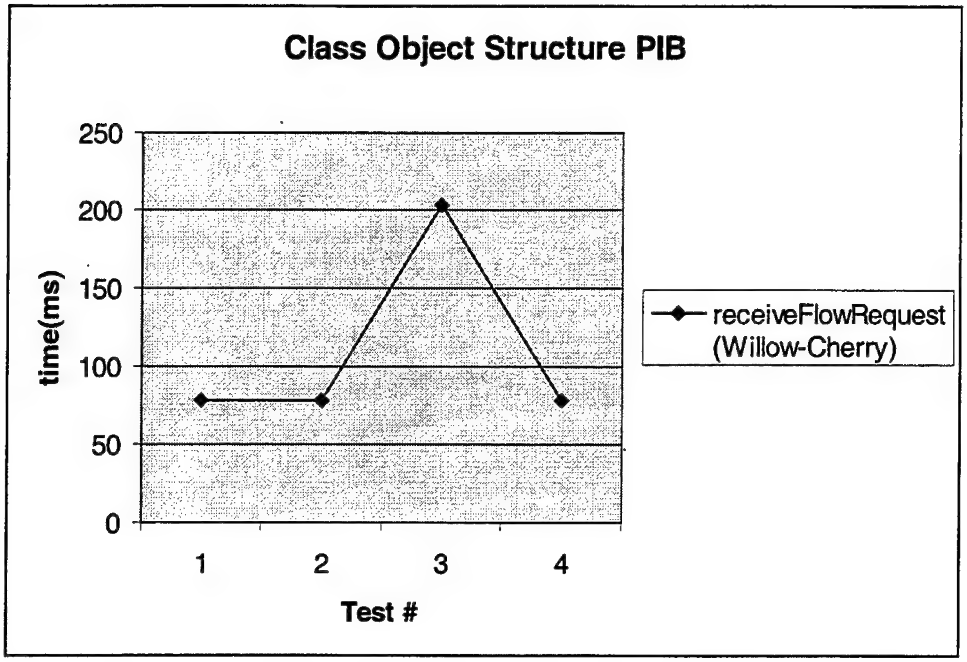


Figure 56. Performance of the Class Object Structure PIB With User Flows

3. Performance Comparison

The performance of the server during this first phase varied greatly depending on whether the PIB was instantiated to use a database or class objects. When instantiated to use a PIB that uses a relational database, the server lagged significantly behind. This significant lag in performance was expected since most relational databases provide persistence by storing and retrieving data to the physical hard drive. Accessing a hard drive is a significantly more expensive operation than accessing a data structure that is completely resident in random assess memory. As the complexity of the network increasing, the significance of the performance differences can be expected to expand greatly. The database structured PIB was developed primarily as a way to arrive at an efficient model of the data that needs to be maintained within the PIB.

VIII. CONCLUSIONS

The study of the issues involved in providing network QoS, as well as the development and implementation of the model, has led us to numerous conclusions. It has also led to a large number of concepts for developing this model further. These lessons learned and future works items are outlined below.

A. LESSONS LEARNED

Several important facts were learned throughout the conduct of this thesis. These discoveries centered around the design of the server and router. Additional lessons were learned relating to networking and Java issues. All of these are described below.

1. SERVER DESIGN

A number of key points came to light during the development and implementation of the server model. The first was realizing that all server functions are driven by the receipt of messages from routers. The second deals with the advantages of implementing the PIB as a class object structure.

a) Server Functions are Driven by the Receipt of Messages

The receipt of different messages drives the entire operation of the server, as currently defined. The operation of the server that results from the receipt of various messages is summarized below.

(1) Hello Message.

- processing of the Hello message
- determining all the possible paths across the network
- determining the effective QoS that these paths are providing

- assigning of a flow back to the router that sent the hello message

(2) LSA Message.

- processing of the LSA message
- determining the effective QoS of the paths passing over this service level pipe
- checking to ensure that the flows assigned to these paths can still be supported

(3) Flow Request Message.

- processing the Flow Request message
- sending of Flow Routing Table Entry messages to each router in the path, if supportable
- sending of the Flow Response message

(4) Flow Termination Message.

- processing the Flow Termination message
- sending of Flow Routing Table Entry messages to each router in the path to remove entry for this flow

b) Implementation of the PIB

Through the development of this thesis, we demonstrated that the method used to implement the PIB can have an immense impact on the performance of the server as a whole. For example, implementing the PIB as a relational database resulted in an increase in the duration of time to find all the possible paths to approximately n*x milliseconds where n is the number of nodes in the PIB and x is the number of milliseconds required to do the same operation using a class object structure.

2. ROUTER DESIGN

a) Object-Oriented Design

The first major design issue that was brought to light concerning the router involved protocol encapsulation. It quickly became apparent that, in order to perform flow-based routing on the networks that were available to us at the time of this printing, we would need several layers of encapsulation. Java's object-oriented design proved invaluable in assisting us with the rapid development of the objects necessary to emulate a protocol stack.

By creating objects that could act autonomously, we were able to re-use code and quickly and easily perform concept revisions that would be considered major in other languages. Two examples of such revisions are: Switching the underlying transport protocol from UDP to TCP in one day; and, migrating the Graphical User Interface (GUI) from a multitude of Java AWT frames populating the screen to one menu-driven, dynamically updated Java Swing GUI in less than two days.

b) Replacement of Class Loaders

Installing a resident agent using Java involves passing a byte array that represents a class file as a parameter into an object called a class loader. One minor weakness in the Java language is that its primordial class loader (the class loader that is built into the language) does not allow a class object with the same name as a previously loaded class object to be loaded more than once.

This would have all but eliminated the possibility of replacing resident agents, but for the trick we learned from Jason Hunter on page 55 of his Java Servlet Programming book. This involves catching the Linkage Error that is thrown when the class loader attempts to load a duplicate class and instantiating a new class loader to load the replacement class. [15]

c) *Primitive Conversions and Dynamically Extensible Arrays*

Converting primitive data types to arrays and dynamically building and parsing arrays quickly became a tedious task without the aid of an object that provided this functionality. Java provided no obvious solutions to this dilemma, so we developed two classes that proved invaluable in assisting us in the multitude of array manipulation and primitive data-type conversions that were necessary throughout this project. The first was the `Array` class. The Array class has several versions of three static methods: `concat`, `getSubArray`, and `replaceElementAt`.

The concat method allows for the combining of two arrays or of one primitive data type with an array of the same type. This method is called by the `PacketFactory`, for example, to build the byte array that constitutes a packet.

The `getSubArray` method provides a way to extract a sub array from an array of primitive data types, while the `replaceElementAt` method returns an array of primitive types with the primitive value to be added inserted into array at the position designated in the method parameter. This method automatically expands array if the current position is greater than the length of array

3. NETWORK ISSUES

The majority of this thesis dealt with network issues. Two of these issues were particularly important. The first, the benefits of TCP over UDP, dealt with implementation, while the second, introduction of delay, was learned while studying networking models that provide QoS.

a) *The Benefits of TCP over UDP*

This first iteration of the SAAM architecture began with the plan to use UDP as the transport level protocol. While this was in keeping with the use of UDP by existing routing protocols, we suffered significant random packet loss on a reasonably uncongested network. During the integration testing, we observed using tcpdump

numerous stub packets that appeared to correspond with our missing packets. At that time, we chose to convert over to TCP in order to get reliable packet transfers. This conversion was successfully accomplished and no further packet loss was observed.

b) Introduction of Delay

Developing the understanding that delay in a network is introduced in the outbound queues of routers and hosts was a significant step. By understanding this, the modeling of the data that needed to be transmitted to the server was significantly easier and cleaner.

B. FUTURE WORK

This thesis is the initial development effort to provide a working model of the proposed SAAM architecture. As such, only the software architecture and core functionality was developed within the time constraints. There are numerous other functions that can now be implemented that will benefit from this general SAAM architecture. In the sections that follow, these other functions are outlined in order to outline the possibilities and to motivate future development.

1. FAULT MANAGEMENT

Fault management opportunities present themselves in three areas: network fault detection, network fault response, and server fault tolerance. Each of these areas is described below.

a) Network Fault Detection

The question of how the server will become aware of a network fault is not fully developed. Each router monitors its directly connected segments for performance and therefore for faults. They communicate their observations to the server via LSA messages. This system is sufficient if it were not for the complexities of our networks. In a shared media environment, such as Ethernet, routers do not currently identify the failure

of another router on the same network. This added functionality would result in the addition of certain data structures at the router and in the modification of the LSA message. The processing of the LSA messages at the server would also need to be modified slightly.

At the server, detection of a network fault could also be accomplished by simply routinely checking to see when the last LSA was sent from each known router. When a router's status update is late, the server could resort to sending a specific LSA request to that router.

b) Network Fault Response

Once network faults or performance degradations are identified, the server must take the steps needed to modify the flows that traverse the affected segments of the network. This response might be as simple as finding another path from the original source to the final destination of the flow. Another solution would be to identify the edges of the network that surround the fault for the affected flows and then find an alternate route around the affected faulty area. This second solution would result in fewer network messages and a possibly faster resolution for the affected flows, at the price of additional processing at the server.

c) Server Fault Tolerance

The failure of a region's SAAM server would end its ability to guarantee any new flows of a particular QoS level. Furthermore, it would also result in existing flows from ever being deleted from the flow routing tables of routers in the region. There are a number of possible solutions to prevent this. First, a redundant server could be implemented on a different host or router. This secondary server would receive PIB updates as the primary server receives them. Since they receive identical input, these servers would produce the same results. The failure of the primary would result in the immediate assumption of primary server functionality by the secondary. Another solution would be to add the software needed for the routers of the region to elect a new router to host a new server and for each router to send their current state.

2. NETWORK CHANGE MANAGEMENT

The subject of managing the response to changes in the network came to the forefront at numerous times during the development of the SAAM model. The first issue in this area occurs at the server. This issue is: as new hello messages arrive, how much time should pass before the PIB is rebuild? The second issue is in this area occurs at the router. This issue is: how much change must occur before an LSA is tranmitted to the server.

a) Frequency Of Physical Network Changes

At network initialization and at times of reconfiguration of the existing physical network, the frequency of how often to rebuild the PIB must be addressed. In a controlled environment, one can send all of the hello messages at once and then have the server take the time to find all of the possible paths across the network in its entirety. In a real network, however, new hello messages could be sent at different frequencies and could be intermixed with flow requests. Should the PIB be rebuilt immediately after receiving a new hello message? Or, should there be some form of dampening of the frequency of these rebuilds can consider a possibly larger number of physical network changes before going through the relatively lengthy process for searching this new network for different paths?

b) Frequency Of LSA Transmissions

The question of how often an LSA message should be sent from a router to the server is still not clearly defined. An LSA message would need to be sent at a certain frequency even if there is no change in its previously reported status. This will simply require the picking of an wise frequency. A more complicated question is how much change from the previously reported status is significant enough to trigger the sending of a new LSA.

3. SERVER HEIRARCHY MANAGEMENT

The management of the server heirarchy consists of giving each server the ability to act like a router. This ability to act like a router entails summarizing the topology and network performance of its region. This process is a scaled up version of what a basic router does to access the possible paths across its internal switching matrix. Furthermore, acting like a router also encompasses the management of those data structures that a router maintains, such as a flow routing table. Both of these issues that are involved in acting as a router are described below.

a) Regional Summarization

As the hierarchy of the SAAM architecture is extended up beyond a single region, the region's SAAM server must summarize the status of the region. This summarization results in the next higher level server simply viewing the region as a router whose status is presented by the hello and LSA messages sent by the region's SAAM server. This summarization process still needs to be developed. The development of the hello message would involve the server identifying its border routers. The development of LSA messages would require the determination of the average QoS experienced by packet that traverses its region. This determination of QoS characteristics might simply consist of determining the average of the QoS characteristics that has been calculated for different paths that connect the border routers.

b) Server Flow Routing Tables

For a server to act as a router, it must also maintain a flow routing table. This flow routing table would consist of entries for flows crossing to neighboring regions. If the source application requesting a flow across regions were resident on a router or host within its region, the higher level server would need to receive a flow request from the lower level server acting as a router. This lower level router would need to substitute itself as the source of the flow request. Once it receive the flow response back from the higher level, the server would be responsible for choosing the best path from the original

source router to the border router that is assigned as the source interface by the higher level server.

4. FLOW MANAGEMENT

There are many complexities involved in the management of flows. Two of the more interesting ones include determining with a flow is terminated and responding to varying need for bandwidth. These two issues are describe in more detail below.[15]

a) Flow Termination

There is currently no methodology developed for the termination of a flow. The simple solution would involve the use of known duration flows. The duration of the flow would be stated during the flow request. The server would then need to remove the flow routing table entries at the end of reservation period.

A slightly more complex and dynamic solution would be to depend on the QoS application to notify the server of the termination of a flow. The dependency introduced by this solution would require active management to ensure that flows are not terminated without notification. A commonly occurring system lock-up could easily produce an unterminated flow. If the source machine might not have retained its state before its lock up, it would have no knowledge of previously assigned flows and would never send flow termination messages.

b) Flow Expected Bandwidth Utilization

The current assumption that has been used in the development of this model so far has been that the flows are constant bandwidth flows. This assumption may not always be true. Accommodations for variability in bandwidth consumption would provide more flexibility to the system.

[15] These issues may not be critical because the SAAM server makes admission critical decisions based on observed network performance.

5. SECURITY MANAGEMENT

As stated in the overview chapter, the issue of ensuring security has yet to be integrated into this development effort. Areas were security is required include authenticating the identity of the senders of network status updates, as well as the sender of flow routing table entries. The integrity of the messages sent between the routers and servers will also need to be protected. There are numerous possible solutions to these issues. Some of the solutions are even being made available by Sun in their continuing development of Java.

6. RESIDENT AGENT MANAGEMENT

While the deployment of resident agents occurred during this development effort, there are a number of ways that we would encourage others to expand on their use in the architecture. These ways include deploying resident agents from the server, creating new resident agents to add functionality to the router, and to extend the resident agent concept to mobile agents.

a) Server Deployment of Resident Agents

The SAAM architecture is designed to allow the dynamic updating of router software from the server through the use of the resident agent concept. While the router is currently composed of objects that are implemented as resident agents, the send and receiving of these resident agents still needs to be developed further. Specifically, the server will need to implement the interfaces to allow the loading and sending of these resident agents out to the routers.

b) Creating New Resident Agents

Many new resident agents could be created to operate within this architecture. For example, a group of Scheduler classes could be developed that would assist the server in managing network traffic.

c) *Mobile Agents*

Since agents can be instantiated temporarily on a router, and then transported between hosts on a network, it makes sense that resident agents could be developed into mobile agents that forward themselves from router to router. A server mobile monitor agent, for example, could be deployed from the server that visits each hop assigned to a given flow, and monitors the link state of that hop. It could then either forward the data to the server, or collects it by forwarding itself to the next hop and then passing its state in the form of a message to that agent. Minor modifications of the `PacketFactory` and `ControlExecutive` classes would be necessary to allow this type of transfer.

7. POLICY MANAGEMENT

The Control Executive receives all resident agents destined for any of the interfaces the Control Executive instantiates. Before instantiating these agents, the Control Executive could be programmed to perform various policy adherence checks such as disallowing agents whose package name is `saam.control` for example. A check such as this would prevent agents from accessing the `Channel` class directly and circumventing the `Channel` registration process.

8. IMPROVEMENTS TO EXISTING CODE

As the implementation of the model progressed and our understanding of the issues involved matured, we observed that we could have implemented some methods in a better or more complete way. In this section, we describe these areas where we would encourage improvements to our existing code.

a) *Search For Paths without Considering Service Levels*

The Server's `findAllPossiblePaths()` currently develops a list of parents for each router at each service level. These lists of parents are identical for each

service level for the same router. A significant amount of time could be saved during this building process if it was reworked to find all possible paths at one service level and then duplicate these paths as new paths at the other service levels without repeating the same search process.

b) Find an Algorithm to Reduce the Path Search Process

When one node or a new interface is added to a network, there may be some sort of algorithm capable of limiting the scope the search that needs to be conducted. This would be particularly advantageous when the network is already sizeable.

c) Develop a Graphical User Interface for the Server

The server is the perfect place to gain a full understanding of the current conditions of the network. An improved GUI for the server could take advantage of the fact. The functions that could be made available to the SAAM system administrator includes the following:

- Configure
    - H_{max}, The Maximum Number of Hop in the Paths Described in the PIB
    - Service Level Configuration for the Region
- Operate
    - Rebuild the Server's Path Information Base
    - Request a Link State Advertisement from a Router
- Display
    - All Routers in the Region
    - All Links in the Region
    - All Flows in the Region
    - Flows Through a Particular Router
    - Percentage of Bandwidth Assigned on an Interface or Service Level Pipe
    - Number of Flows Originating or Terminating at a Particular Router
    - Where the Maximum Delay or Loss is Occurring in the Network
    - The Last Link State Advertisement from a Router
    - All Routers Connected to a Particular Link

d) Passing Messages vice Events on Channels

Channels are currently written to accept only event objects as parameters. Since event objects are designed to convey information from one object to another, it makes sense to craft the Message class in such a way that it can be passed between objects as well. By passing messages instead of event objects, we could merge most of the classes within the event package with those in the message package.

e) Empowering the Control Executive

Through the use of the reflection API, that is, the classes in the package java.lang.reflect, the Control Executive could be empowered to "reflect" into the bytecode of resident agents and determine many things about them.

f) Naming and Numbering Channels

Some convention should be adopted for naming and numbering Channels. In our design, the Channels numbered 0-[max port number] were designated as the Channels used to emulate ports; and the numbers above max port number were designated as the Channels used for other purposes.

To name our Channels, we would typically use the following format:

FROM_<object>_TO_<object>_CHANNEL for objects in which only one Channel was required for all instances; and

FROM_<object>_TO_<object>_START_CHANNEL for objects in which several Channels were required between instances. This format was always accompanied by an accessor method which, when supplied with an integer representing the instance number of the latter object, would return the Channel associated with that instance. The accessor methods were in the following format:

getFrom<object>To<object>StartChannel(int instanceNumber)

APPENDIX A. SAAM PACKAGE SOURCE CODE

```
package saam;

import saam.net.*;
import saam.util.Array;

/**
 * An EmulationPacket object contains a header
 * and a payload.  The header contains an eight-byte time stamp
 * and a one-byte numberOfUpdates field.  The payload should consist
 * of a series of entries as follows:<p>
 *   A one-byte type field identifying a Message
 *   Followed by a series of bytes representing the entry of
 *   the type identified.
 */
public class EmulationPacket {

  /**
   * This variable defines the number of bytes in the
   * numberOfUpdates field
   */
  //the numberOfUpdates field is defined in this class.
  public static final int numberOfUpdatesLength = 1;

  /**
   * The header field for this EmulationPacket.
   */
  private byte numberOfUpdates = 1;

  /**
   * The payload field for this EmulationPacket.
   */
  private byte[] payload;

  /**
   * The byte array representation of this packet.
   */
  private byte[] packet;


    /**
     * Constructs an EmulationPacket given appropriately
     * defined header (numberOfUpdates) and payload fields.
     * @param numberOfUpdates The number of updates that are
     *                        included in this packet.
     * @param payload The payload of this packet.
     */
    EmulationPacket(byte numberOfUpdates, byte[] payload) {
        this.numberOfUpdates = numberOfUpdates;
        this.payload = payload;

        //populate the packet array
        packet = Array.concat(packet, numberOfUpdates);
        packet = Array.concat(packet, payload);
    }

    /**
     * Constructs an EmulationPacket from a byte array.
     */
    EmulationPacket(byte[] packet){
      this.packet=packet;
      this.numberOfUpdates = packet[0];
```

```
    payload = Array.getSubArray(packet,numberOfUpdatesLength,
      packet.length);
  }

  /**
   * Returns the numberOfUpdates in the EmulationPacket header.
   * @return The numberOfUpdates in the EmulationPacket header.
   */
  public byte getNumberOfUpdates(){
    return numberOfUpdates;
  }

  /**
   * Returns the payload as a byte array.
   * @return The payload as a byte array.
   */
  public byte[] getPayload(){
    return payload;
  }

  /**
   * Returns the EmulationPacket as a byte array.
   * @return The EmulationPacket as a byte array.
   */
  public byte[] getPacket(){
    return packet;
  }
}
```

```
package saam;

import java.util.Hashtable;
import java.util.Vector;
import java.util.Enumeration;
import java.net.InetAddress;
import java.net.UnknownHostException;
import saam.net.IPv6Address;
import saam.message.*;

import saam.util.TableGui;

/**
 * The <em>emulation table</em> stores the IPv4 addresses
 * associated with a given IPv6 addresses.  This enables the
 * <code>Translator</code> to pass emulated SAAM traffic
 * over an existing IPv4 network.<p>
 */
public class EmulationTable extends Hashtable{

  private TableGui gui;
  private Vector index=new Vector();
  private Vector names = new Vector();
  /**
   * The constructor creates an emulationTable with initial
       * capacity of capacity.  Initial capacity should be a
       * prime number to help distribute the entries evenly
       * among the hash buckets.
       */
      public EmulationTable(int capacity){
    super(capacity, 0.5f);
    names.add("IPv6 Address");
    names.add("IPv4 Address");
    int[] columnWidths = {260,120};
    gui = new TableGui(toString(), names, columnWidths);
      }

  /**
   * Creates a new entry in the <em>emulation table</em>.
   * @param entry The EmulationTableEntry to be added to the
EmulationTable.
   */
  public void add(EmulationTableEntry entry){
     IPv6Address v6Addr = entry.getIPv6Address();
     String v6 = v6Addr.toString();
     String v4 = entry.getIPv4Address().getHostAddress();
     put(v6,entry);
     gui.fillTable(getTable());
  }

      /**
       * Removes an entry from the <em>emulation table</em>.
   * @param entry The EmulationTableEntry to be removed from the
EmulationTable.
       */
      public void remove(EmulationTableEntry entry){
    String v6 = entry.getIPv6Address().toString();
            remove(v6);
    gui.fillTable(getTable());
      }
```

```
    /**
     * Retrieves an EmulationTableEntry from the emulation table.
     * @param v6 The IPv6 address to be used as the lookup key.
     * @return The EmulationTableEntry associated with v6.
     */
    public EmulationTableEntry get(IPv6Address v6){
          return (EmulationTableEntry) get(v6.toString());
    }

  /**
   * Returns the entire contents of this EmulationTable or null
   * if this EmulationTable is empty.
   * @return An array of all emulation table entries currently
   *         in the emulation table.
   */
  public Vector getTable(){
    if(isEmpty()) return null;
    Vector table = new Vector(size());
    for(Enumeration e = elements();
        e.hasMoreElements();){
      Vector oneRow = new Vector();
      EmulationTableEntry entry = (EmulationTableEntry)e.nextElement();
      oneRow.add(entry.getIPv6Address().toString());
      oneRow.add(entry.getIPv4Address().getHostAddress());
      table.add(oneRow);
    }//for
    return table;
  }//getEmulationTable()

  /**
   * Returns the String representation of this Object.
   * @return The String representation of this Object.
   */
  public String toString() {
      return "Emulation Table";
  }//toString()
}
```

```
package saam;

import java.net.ServerSocket;
import java.net.SocketException;
import java.net.Socket;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.TooManyListenersException;
import java.util.Vector;
import java.util.Hashtable;
import java.io.IOException;
import java.io.OutputStream;
import java.io.InputStream;

import saam.message.*;
import saam.router.*;
import saam.util.SAAMRouterGui;
import saam.util.Array;
import saam.residentagent.ResidentAgent;
import saam.message.MessageProcessor;
import saam.event.*;
import saam.net.*;
import saam.control.*;

/**
 * The <em>translator</em> is not a part of the <em>SAAM</em>
 * architecture.  Rather, it is an aide to demonstrate the
 * architecture's proof of concept.  The purpose of the
 * <em>translator</em> is to perform the conversions needed
 * for the <em>SAAM Router</em> to operate in an IPv4 environment.
 * The <em>translator</em> has an <em>Emulation Table</em> which
 * contains the necessary conversion information.  This EmulationTable
is
 * updated when an EmulationTableEntry Message arrives.  Since the
Translator
 * is a MessageProcessor, it is allowed to register with the
ControlExecutive to
 * receive saam.message.EmulationTableEntry messages.<p>
 * When instantiated, the <em>translator</em> first stands up a
ControlExecutive
 * and then listens on port 9001 for emulation traffic from the
DemoStation.
 * When emulation packets are received on port 9001, they are forwarded
on the
 * PacketFactory Channel for processing.
 * <p>Additionally, the Translator registers to listen on the
ROUTER_STATUS_CHANNEL
 * which is used by the ControlExecutive to notify the Translator that
all
 * elements necessary to stand up a router have been received.  Once the
Translator
 * receives this notification from the ControlExecutive, the Translator
stands up a
 * PortListener on the SAAM port (9002).  When packets are received on
the SAAM port,
 * they are forwarded to the NetworkInterfaceCard objects on the
 * FROM_TRANSLATOR_TO_NICS_CHANNEL.
 * @see  saam.control.PacketFactory#receiveEvent(SaamEvent se)
 * @see  saam.control.ControlExecutive#updateRouterStatus()
 */
```

```java
public class Translator extends Thread implements
  MessageProcessor, SaamTalker, SaamListener{

  //the types of messages to be processed by the Translator
  private static final String[] messageTypes =
    {"saam.message.EmulationTableEntry"};

  private ControlExecutive controlExec;

  private Hashtable openConnections = new Hashtable();
  private PortListener saamPortListener;
  private PortListener emulationPortListener;
  private EmulationTable emulationTable;
  private ServerSocket emulationSocket,saamSocket,outboundSocket;
  private SAAMRouterGui gui;
  private boolean routerStarted = false;
  private int saamPort, emulationPort;
  private int maxSAAMPacketSize, maxEmulationPacketSize;
  private int actualPacketSize;
  private int outboundPacketCount = 0;


  /**
   * Since the Translator is the main class file, we include
   * a main method to enable command-line instantiation.
   */
  public static void main(String args[]){
    try{
        Translator t = new Translator(9002,9001);
    }catch(IOException ioe){
      System.out.println(ioe.toString());
    }//try-catch
  }//main

 /**
 * Constructs a new <code>Translator</code> listening on the
 * <em>saamPort</em> and <em>emulationPort</em> provided for packets
 * no larger than the <em> maxSAAMPacketSize</em> provided.<p>
 * Here, the Translator instantiates a ControlExecutive, and then
 * registers with that ControlExecutive to talk on the channel that
 * passes traffic from the Translator to the NetworkInterfaceCard
 * objects.  The Translator then registers with the ControlExecutive
 * to listen on the Channel that passes traffic in the opposite
 * direction; and, on the ROUTER_STATUS_CHANNEL which is used by the
 * ControlExecutive to notify the Translator that all elements
 * necessary to stand up a router have been received.
 * @see  saam.control.ControlExecutive#updateRouterStatus()
 * @param saamPort The saamPort which this <em>translator</em> will
 * listen on.
 * @param maxSAAMPacketSize The maximum size of an incoming SAAM packet.
 * @param emulationPort The emulationPort which this <em>translator</em>
 * will listen on.
 * @param maxEmulationPacketSize The maximum size of an incoming
 * emulation packet.
 */
 public Translator(int emulationPort, int saamPort)
        throws IOException{
    gui = new SAAMRouterGui(toString());
    try{
      gui.setTextField("My IP: "+
        InetAddress.getLocalHost().getHostAddress());
```

```
    }catch(UnknownHostException uhe){}
    controlExec =
      new ControlExecutive();

    //*****************************************************/
    //Enable talking
    //*****************************************************/
    int channel = ProtocolStackEvent.FROM_TRANSLATOR_TO_NICS_CHANNEL;
    try{
       controlExec.addTalkerToChannel(this,
          channel);
    }catch(ChannelException ce){
    }

    //*****************************************************/
    //Enable listening
    //*****************************************************/
    channel = ProtocolStackEvent.FROM_NICS_TO_TRANSLATOR_CHANNEL;

        try{
           controlExec.addListenerToChannel(this, channel);
        }catch(ChannelException ce){
        }//try-catch

        channel = controlExec.ROUTER_STATUS_CHANNEL;
        try{
           controlExec.addListenerToChannel(this, channel);
        }catch(ChannelException ce){
        }//try-catch

        //the emulationTable for this Translator is stood up next
    emulationTable = new EmulationTable(1711);

    this.saamPort = saamPort;
    this.maxSAAMPacketSize =  maxSAAMPacketSize;

    this.emulationPort = emulationPort;
    this.maxEmulationPacketSize =  maxEmulationPacketSize;

    //the channel on which the emulationPortListener will communicate
    channel = ProtocolStackEvent.PACKETFACTORY_CHANNEL;

    //the ServerSocket the emulationPortListener will listen to
    emulationSocket = new ServerSocket(emulationPort);
    emulationPortListener = new PortListener(emulationSocket,channel);
    controlExec.registerMessageProcessor(this);
    start();
  }//Translator()
  public void run(){
  }

  /**
   * Returns the array of Strings representing the class
   * names of the messages this Object will register to process.
   * @return The array of Strings representing the class names
   * of the messages this Object will register to process.
   */
  public String[] getMessageTypes(){
    return messageTypes;
  }
```

```java
/**
 * This method contains the logic needed by the Translator
 * to process EmulationTableEntry objects.  These table
 * entries can be additions to the EmulationTable or deletions
 * from it (determined by the length of the EmulationTableEntry).
 */
public synchronized void processMessage(Message message){
  //get the class name of the Message to be sure it's of the type
  //the Translator registered for.
  String name = message.getClass().getName();
  gui.sendText("Got a Message:\n"+name);
  if(name.equals(messageTypes[0])){
    //flowZeroNextHop is the String representation of the IPv6Address
    //of the next hop associated with flow zero.  We use it here to
    //notify the ControlExecutive when the EmulationTableEntry
    //containing
    //this address arrives.
    String flowZeroNextHop =
      controlExec.getFlowZeroNextHop().toString();
    EmulationTableEntry entry = (EmulationTableEntry)message;
    if(message.length()==IPv6Address.length+IPv4Address.length){
      //add this record
          emulationTable.add((EmulationTableEntry)message);
    }else {
      //remove this record
          emulationTable.remove((EmulationTableEntry)message);
    }
    if(!controlExec.getEmulationTableStatus() &&
       !flowZeroNextHop.equals(IPv6Address.DEFAULT_HOST)){
      if(emulationTable.containsKey(flowZeroNextHop)){
        controlExec.updateCoreServiceStatus(this, true);
      }else{
        controlExec.updateCoreServiceStatus(this, false);
      }
    }
  }
}

/**
 * Receives and processes the SaamEvent notification
 * generated by the packet talker to whom this Translator is
 * listening.
 * @param pe The SaamEvent to be processed.
 */
public synchronized void receiveEvent(SaamEvent se){
  int channel = se.getChannel_ID();
  if(channel == ControlExecutive.ROUTER_STATUS_CHANNEL){
    if (((RouterStatusEvent)se).getStatus()==true){
          try{
        gui.sendText("Standing up SaamPort: "+saamPort);

        //the DatagramSocket the saamPortListener will listen to
        saamSocket = new ServerSocket(saamPort);

        //the channel on which the saamPortListener will communicate
        int channelToNics =
          ProtocolStackEvent.FROM_TRANSLATOR_TO_NICS_CHANNEL;

        saamPortListener =
          new PortListener(saamSocket, channelToNics);
          }catch(IOException ioe){
```

```
            gui.sendText(ioe.toString());
          }
        }else{
          //future work...
          //  shutDownSaamPort();
        }
      }else{
            ProtocolStackEvent pse = (ProtocolStackEvent)se;
            if(channel ==
ProtocolStackEvent.FROM_NICS_TO_TRANSLATOR_CHANNEL){
          gui.sendText("Got an outbound packet...");
          byte[] packet = pse.getPacket();
          //outbound packet
          outboundPacketCount++;

          IPv6Address nextHop = null;
          try{
            nextHop = new IPv6Address(
              Array.getSubArray(packet,0,IPv6Address.length));
          }catch(UnknownHostException uhe){
            gui.sendText(uhe.toString());
          }
          gui.sendText("    Next hop:" + nextHop.toString());
 /*
          // print more debugging information
          IPv6Packet v6Packet = null;
          try{
            v6Packet = new IPv6Packet(Array.getSubArray(packet,
IPv6Address.length+1, packet.length));
          }catch(UnknownHostException uhe){
            gui.sendText("Couldn't instantiate IPv6Packet");
          }
          gui.sendText("    Source: " +
v6Packet.getHeader().getSource().toString());
          gui.sendText("    Destin: " +
v6Packet.getHeader().getDest().toString());
          gui.sendText("    Payload length = " +
v6Packet.getPayload().length +
                      "; Flow label = " +
v6Packet.getHeader().getFlowLabel());
*/
          InetAddress nextPhysicalHop = null;
          try{
            nextPhysicalHop =
            ((EmulationTableEntry)emulationTable.get(nextHop)).
              getIPv4Address();
          }catch(NullPointerException exc){
            gui.sendText(exc.toString());
          }
          packet = Array.getSubArray(packet,IPv6Address.length,
            packet.length);
          gui.sendText("Sending outbound packet to: "+
          nextPhysicalHop.getHostAddress());
          gui.sendText("DestPort: "+saamPort);
          try{
            Socket sock =

(Socket)openConnections.get(nextPhysicalHop.getHostAddress());
            OutputStream out = sock.getOutputStream();
            out.write(packet);
            out.flush();
```

```
          out.close();
        }catch(NullPointerException npe){
          //put the Socket into the Hashtable for outbound
          //packets destined for this socket.
          try{
            Socket sock = new Socket(nextPhysicalHop,saamPort);
            openConnections.put(
              nextPhysicalHop.getHostAddress(), sock);
            OutputStream out = sock.getOutputStream();
            out.write(packet);
            out.flush();
            out.close();
          }catch(IOException ioe){
            gui.sendText(ioe.toString());
          }
        }catch(IOException ioe){
          gui.sendText(ioe.toString());
        }
          }//if
    }//else
  }//receiveEvent

    /**
     * Returns a <code>String</code> representation of this object
     * @return The <code>String</code> representation of this object
     */
  public String toString(){
      return "Translator";
  }

  /**
   * A threaded inner class used by the <code>Translator</code> to
   * receive and forward <code>DatagramPackets</code>.
   */
  class PortListener implements SaamTalker,Runnable{

      private ServerSocket serverSocket;
      private int channel;
  private String name;
  private Socket sock;

      /**
       * Constructs a new <code>PortListener</code> and a
<code>Thread</code>
   * for the listener; and then starts the <code>Thread</code>.
       * @param socket The <code>DatagramSocket</code> the PortListener
       * will receive packets on.
       * @param maxPacketSize The maximum size an inbound packet can
have.
   * @param channel The channel on which the inbound packet will be
   * communicated.
       */
      PortListener(ServerSocket serverSocket, int channel){

     //***************
     //Enable talking
     //***************/
     try{
       controlExec.addTalkerToChannel(this,channel);
     }catch(ChannelException ce){
       gui.sendText(ce.toString());
```

```
    }
            this.serverSocket=serverSocket;
            this.channel=channel;
    name = "portListener("+channel+")";
            Thread listenerThread = new Thread(this, name);
            listenerThread.start();
      }

      /**
       * On a separate <em>thread</em>, this method listens for packets.
       * When a packet is received, its exact size is determined, then
the
       * listeners are notified of the packets arrival.
       */
      public void run(){
    long packetCounter = 0;
            gui.sendText(toString()+": now listening on port: "+
                        serverSocket.getLocalPort());
    // wait for packet
    while (true) {
        try {
            sock = serverSocket.accept();
            sock.setReceiveBufferSize(65508);
        } catch (SocketException se){
            gui.sendText(se.toString());
        } catch (IOException e) {
            gui.sendText(e.toString());
        }
        //here we declare an annonymous inner class that
        //listens to the socket on a separate thread.
        (new Runnable(){

            byte[] buf = new byte[65508];
            int c;
            int i=0;
            public void run(){
               //while(true){
                 InputStream is = null;
                 try{
                    is = sock.getInputStream();
                    InetAddress source = sock.getInetAddress();
                     int j = 0;
                     while ((c=is.read()) > -1) {
                       //if (c>0)
                       if (i < buf.length)
                         buf[i++] = (byte)c;
                       else
                         i = 0;
                     }
                     byte[] packet = Array.getSubArray(buf,0,i);
                     gui.sendText(">>> Got an inbound packet");
                     gui.sendText("    Total length = " +
packet.length);
 /*
                     // print more debugging information
                     IPv6Packet v6Packet = null;
                     try{
                       v6Packet = new
IPv6Packet(Array.getSubArray(packet,1,packet.length));
                     }catch(UnknownHostException uhe){
```

```
                        gui.sendText("Couldn't instantiate IPv6Packet");
                      }
                      gui.sendText("    Source: " +
v6Packet.getHeader().getSource().toString());
                      gui.sendText("    Destin: " +
v6Packet.getHeader().getDest().toString());
                      gui.sendText("    Payload length = " +
v6Packet.getPayload().length +
                                   "; Flow label = " +
v6Packet.getHeader().getFlowLabel());
 */

                      //now forward the packet on the appropriate
Channel.
                      forwardPacket(source,packet);
                 }catch(IOException ioe){
                   gui.sendText(ioe.toString());
                 }//try-catch
               //}//while(true)
             }//run()
         }).run();
     }//while()
  }//run()

  private synchronized void forwardPacket(InetAddress source,byte[]
packet){
     ProtocolStackEvent event = new ProtocolStackEvent(
     source.getHostAddress(),
     this,
     channel,
     packet);
     try{
         gui.sendText("Sending Notification on Channel "+channel);
         controlExec.talk(event);
     }catch(ChannelException tde){
       gui.sendText(tde.toString());
     }
}

  /**
   * Returns a <code>String</code> representation of this object
   * @return The <code>String</code> representation of this object
   */
  public String toString(){
    return "Translator$"+name;
  }

  }//PortListener
}//Translator
```

APPENDIX B. CONTROL PACKAGE SOURCE CODE

```
package saam.control;

import java.util.Vector;
import java.util.TooManyListenersException;
import java.util.Enumeration;

import saam.event.*;

/**
 * A Channel is a means for one or more objects within the router
 * to communicate with one or more other objects within the router.
 * The ControlExecutive controls access to Channels.  Objects register
 * as talkers or listeners on channels via the ControlExecutive.<p>
 * By registering to communicate on a Channel, Objects are able to
 * set up communications with other Objects that may or may not be
 * instantiated or to Objects that may not yet exist.
 */
class Channel
  //extends Thread
  {

  /**
   * The time this Channel was instantiated
   */
  private long timeActivated;

  /**
   * The time this Channel was last used
   */
  private long timeLastUsed;

  /**
   * The integer identifying this Channel
   */
  private int channel_ID;

  /**
   * The Vector of objects registered to talk on this Channel
   */
  private Vector talkers = new Vector();

  /**
   * The Vector of objects registered to listen on this Channel
   */
  private Vector listeners = new Vector();

  /**
   * boolean value used to determine the running status of the
   * Thread.
   */
  private boolean running=false;
  private boolean firstEvent = true;

  /**
   * The event that will be spoken on this Channel.
   */
//  private SaamEvent event;

  /**
   * A simple counter for the events communicated on this Channel
   */
```

```
  private int eventsSpoken;

  /**
   * The owner of the current Thread.
   */
  private Thread owner;
  /**
   * This simple run method waits until a talker speaks an
   * event on this Channel.  Once an event is spoken, the
   * notifyListeners method is called.
   */
/*  public void run(){
    while(true){
      try{
        if(!running){
          synchronized(this){
            while(!running) wait();
            running=true;
          }//synchronized
        }
      }catch(InterruptedException ie){
        //do something with this
        System.out.println("Channel **INTERRUPTED!** "+ie.toString());
      }//try-catch
      notifyListeners();
    }//while(running)
  }//run()
*/
  /**
   * This method is not accessible to Objects outside the saam.control
   * package.  It is called by the ControlExecutive when an Object that
   * is registered to talk on this Channel attempts to do so.
   * @param event The event that will be passed to the listeners
   * on this Channel.
   */
  synchronized void talk(SaamEvent event){

    Vector copy = null;
    //synchronized(listeners){
      copy = (Vector)listeners.clone();
    //}
    eventsSpoken++;
    Enumeration e = copy.elements();
    while(e.hasMoreElements()) {
      SaamListener listener = (SaamListener)e.nextElement();
      // print debugging information
      // System.out.println("Notifying: "+listener);
      // System.out.println("Event: "+event);
      listener.receiveEvent(event);
    }//while

/////////////////////////////////////////////////////
/*    if(this.event==null){
      this.event=event;
    }else{
      synchronized(this.event){
        this.event = event;
      }
    }
    //see run()
    if(!firstEvent){
```

```
        if(!running){
          running=true;
          notify();
        }else{
          notifyListeners();
        }
      }else{
        firstEvent=false;
        running=true;
        start();
      }
      event=null;
*/
  }//talk()

  /**
   * This method iterates through the Vector of registered listeners
   * and calls the receiveEvent method on each.
   */
/*  private void notifyListeners(){
    //at this point, we know that the correct PacketTalker thread
    // is running
    //clone the Vector
    Vector copy = null;
    //synchronized(listeners){
      copy = (Vector)listeners.clone();
    //}
    eventsSpoken++;
    Enumeration e = copy.elements();
    while(e.hasMoreElements()) {
      SaamListener listener = (SaamListener)e.nextElement();
      listener.receiveEvent(event);
    }//while
    running=false;
  }//notifyListeners()
*/
  /**
   * Not accessible to Objects outside the saam.control package.
   * Constructs a Channel with one talker and no listeners.
   * @param channel_ID The integer identifying this Channel
   * @param talker The talker that has registered with the
ControlExecutive.
   */
  Channel(int channel_ID, SaamTalker talker){
    this.channel_ID = channel_ID;
    timeActivated = System.currentTimeMillis();
    timeLastUsed = timeActivated;
    talkers.add(talker);
//    start();
  }//Channel()

  /**
   * Not accessible to Objects outside the saam.control package.
   * Constructs a Channel with one listener and no talkers.
   * @param channel_ID The integer identifying this Channel
   * @param listener The listener that has registered with the
   * ControlExecutive.
   */
  Channel(int channel_ID, SaamListener listener){
    this.channel_ID = channel_ID;
    timeActivated = System.currentTimeMillis();
```

```
    timeLastUsed = timeActivated;
    listeners.add(listener);
//    start();
  }//Channel()

  Vector getChannel(){
    Vector channel = new Vector();
    int talkerCount=0;
    int listenerCount=0;
    try{
      talkerCount = talkers.size();
    }catch(NullPointerException npe){
      talkerCount=0;
    }
    try{
      listenerCount = listeners.size();
    }catch(NullPointerException npe){
      listenerCount=0;
    }
    int max = (talkerCount>=listenerCount?talkerCount:listenerCount);
    int min = (talkerCount<listenerCount?talkerCount:listenerCount);

    for(int i=0;i<min;i++){
      Vector oneRow = new Vector();
      oneRow.add(""+channel_ID);
      oneRow.add((SaamTalker)talkers.get(i));
      oneRow.add((SaamListener)listeners.get(i));
      channel.add(oneRow);
    }//for
    for(int i=min;i<max;i++){
      Vector oneRow = new Vector();
      oneRow.add(""+channel_ID);
      if(talkerCount==max){
        oneRow.add((SaamTalker)talkers.get(i));
      }else{
        oneRow.add("...");
      }
      if(listenerCount==max){
        oneRow.add((SaamListener)listeners.get(i));
      }else{
        oneRow.add("...");
      }
      channel.add(oneRow);
    }
    return channel;
  }

  /**
   * Returns a Vector representing the headers for the columns of a
   * table this data might be placed in.
   * @return The Vector representing the headers for the columns of a
   * table this data might be placed in.
   */
  public static Vector getColumnHeaders(){
    Vector headers = new Vector();
    headers.add("Channel");
    headers.add("Talkers");
    headers.add("Listeners");
    return headers;
  }
```

```
public int[] getColumnWidths(){
  int[] widths = {60,200,200};
  return widths;
}
/**
 * Returns a <code>String</code> representation of this object
 * @return The <code>String</code> representation of this object
 */
public String toString(){
  return
    "Talkers: "+talkers.toString()+
    ", Listeners: "+listeners.toString()+
    ", Idle time: "+getIdleTime()+
    ", Events spoken: "+eventsSpoken;
}//displayState()

/**
 * Not accessible to Objects outside the saam.control package.
 * Adds a registered talker to this Channel's Vector of talkers.
 * @param talker The registered talker to be added to the Vector.
 */
synchronized void addTalker(SaamTalker talker){
  if(!talkers.contains(talker)){
    if(talkers.isEmpty()){
      timeActivated = System.currentTimeMillis();
      timeLastUsed = timeActivated;
    }
    talkers.add(talker);
  }
}//addTalker()

/**
 * Not accessible to Objects outside the saam.control package.
 * Adds a registered listener to this Channel's Vector of listeners.
 * @param listener The registered listener to be added to the Vector.
 */
synchronized void addListener(SaamListener listener){
  if(!listeners.contains(listener)){
    listeners.add(listener);
  }
}//addListener()

/**
 * Not accessible to Objects outside the saam.control package.
 * Removes a registered talker from this Channel's Vector of talkers.
 * @param talker The registered talker to be removed from the Vector.
 */
synchronized void removeTalker(SaamTalker talker){
  talkers.remove(talker);
}//removeTalker()

/**
 * Not accessible to Objects outside the saam.control package.
 * Removes a registered listener from this Channel's Vector of
listeners.
 * @param listener The registered listener to be removed from the
Vector.
 */
synchronized void removeListener(SaamListener listener){
  listeners.remove(listener);
}//removeListener()
```

```
   /**
    * Checks to see if talker is in the Vector of talkers for this
 Channel.
    * @param talker The SaamTalker to be used in the query
    * @return True if the talker is in the Vector, false otherwise.
    */
   synchronized boolean isRegistered(SaamTalker talker){
     return talkers.contains(talker);
   }

   /**
    * Checks to see if listener is in the Vector of listeners for this
 Channel.
    * @param listener The SaamTalker to be used in the query
    * @return True if the listener is in the Vector, false otherwise.
    */
   synchronized boolean isRegistered(SaamListener listener){
     return talkers.contains(listener);
   }

   /**
    * Returns the Vector of SaamTalkers for this Channel
    * @return The Vector of SaamTalkers for this Channel
    */
   synchronized Vector getTalkers(){
     return talkers;
   }//getTalkers()

   /**
    * Returns the Vector of SaamListeners for this Channel
    * @return The Vector of SaamListeners for this Channel
    */
   synchronized Vector getListeners(){
     return listeners;
   }//getListeners()

   /**
    * returns a long that represents the time this Channel was
 instantiated.
    * @return A long that represents the time this Channel was
 instantiated.
    */
   synchronized long getTimeActivated(){
     return timeActivated;
   }//getTimeActivated()

   /**
    * returns a long that represents the time this Channel was last used.
    * @return A long that represents the time this Channel was last used.
    */
   synchronized long getTimeLastUsed(){
     return timeLastUsed;
   }//getTimeLastUsed()

   /**
    * Not accessible to Objects outside the saam.control package.
    * Sets the time this Channel was last used.
    */
   synchronized void setTimeLastUsed(){
     timeLastUsed = System.currentTimeMillis();
```

```
  }//setTimeLastUsed()

  /**
   * Determines and returns the amount of idle time for this Channel.
   * @return A long representing the amount of idle time for this
Channel.
   */
  synchronized long getIdleTime(){
    return System.currentTimeMillis() -
      getTimeLastUsed();
  }//getIdleTime()

  /**
   * Determines and returns the amount of time that has passed since
this
   * Channel was instantiated.
   * @return A long representing the amount of time that has passed
since this
   * Channel was instantiated.
   */
  synchronized long getAgeOfLease(){
    return System.currentTimeMillis() -
      getTimeActivated();
  }//getAgeOfLease()

  /**
   * Returns the integer identifying this Channel
   * @return The integer identifying this Channel
   */
  synchronized int getChannel_ID(){
    return channel_ID;
  }//getChannel_ID()

  /**
   * Returns the number of events communicated over this Channel
   */
  synchronized int getEventsSpoken(){
    return eventsSpoken;
  }//getPacketsSpoken()

  /**
   * Returns true if this Channel has any SaamTalkers in its Vector of
talkers, otherwise it
   * returns false.
   */
  synchronized boolean hasTalkers(){
    return (!talkers.isEmpty());
  }//hasTalkers()

  /**
   * Returns true if this Channel has any SaamListeners in its Vector of
listeners, otherwise it
   * returns false.
   */
  synchronized boolean hasListeners(){
    return (!listeners.isEmpty());
  }//hasTalkers()

}//Channel
```

```
package saam.control;

import java.net.UnknownHostException;
import java.net.InetAddress;
import java.util.*;

import saam.Translator;
import saam.router.*;
import saam.net.*;
import saam.message.*;
import saam.residentagent.*;
import saam.residentagent.router.*;
import saam.event.*;
import saam.util.SAAMRouterGui;
import saam.util.Array;

/**
 * The ControlExecutive maintains control over the event handling
mechanism
 * within the saam protocol stack by acting as a registrar for Objects
that
 * wish to communicate on Channels or emulated ports.<p>
 * The ControlExecutive receives all ResidentAgents destined for any of
the
 * router interfaces the ControlExecutive instantiates.  Before
instantiating these
 * agents, the ControlExecutive could be programmed to perform various
policy
 * adherence checks such as disallowing agents whose package name is
 * saam.control for example.  A check such as this would prevent agents
from
 * accessing the Channel class directly and circumventing the Channel
 * registration process.<p>
 * The ControlExecutive passes a copy of itself to each ResidentAgent
that it
 * instantiates, thus allowing the agent access to the
ControlExecutive's
 * public methods.  Through the use of these methods, ResidentAgents can
 * register to talk on or listen to SAAM ports or Channels, request
flows,
 * send Messages or SAAMPackets, register as MessageProcessors, or
retrieve
 * various types of information from the ControlExecutive.<p>
 * The ControlExecutive also receives all Message Objects that are
destined
 * for this router.  A Hashtable of MessageProcessors is maintained to
 * determine which processor to pass an incoming Message to.<p>
 */
public class ControlExecutive
  implements MessageProcessor, SaamTalker, SaamListener{

  //some well-known UDP ports
  public static final int ECHO_PORT          = 7;
  public static final int DISCARD_PORT       = 9;
  public static final int DAYTIME_PORT       = 13;
  public static final int TIME_SERVER_PORT   = 37;
  public static final int DNS_PORT           = 53;
  public static final int WWW_HTTP_PORT      = 80;
  public static final int CHAT_PORT          = 531;

  //saam ports/channels
```

```
  public static final int HIGHEST_WELL_KNOWN_PORT        = 1023;
  public static final int MAX_PORT                       = 65531;
  public static final int SAAM_CONTROL_PORT
          = 8000;
  public static final int ROUTER_STATUS_CHANNEL
      = 80000;

  /**
   * The ControlExecutive registers with itself as a MessageProcessor
capable of
   * processing messages of the following types.
   */
  private static final String[] messageTypes =
    {"saam.message.InterfaceID",
     "saam.message.ServerID",
     "saam.message.FlowResponse",
     "saam.message.DemoHello"};

  private static final boolean ROUTER_UP = true;
  private static final boolean ROUTER_DOWN = false;

  /**
   * The initial status of the router is false.  As key router
components
   * are added, this status is updated to reflect the router's ability
   * to route packets.
   */
  private boolean routerStatus = ROUTER_DOWN;

  /*
   The following boolean variables represent the status of the elements
necessary
   to stand up a router.
  */
  private boolean helloMessageReceived;
  private boolean arpCacheReady;
  private boolean flowRoutingTableReady;
  private boolean emulationTableReady;
  private boolean outboundInterfaceReady;


  private int interfaceCount;
  private int numberOfSchedulersPresent;
  private int nextInboundInterface;
  private IPv6Address flowZeroNextHop = new IPv6Address();
  private SAAMRouterGui gui;
  private MainGui mainGui;
  private PacketFactory packetFactory;
  private TransportInterface transportInterface;
  private ResidentAgent arpCache;
  private RoutingAlgorithm routingAlgorithm;

  private Interface currentInterface;

  /**
   * If a ServerID Message comes from the DemoStation,
   * the IPv6Address associated with that ServerID will
   * be set as the dest in the IPv6Header of packets sent out
   * on flow zero, otherwise, the default IPv6Address
   * will be set as the dest.
   */
```

```
    private IPv6Address serverIP = new IPv6Address();

    /**
     * The ServerID will be sent by the DemoStation.
     */
    private ServerID serverID;

    /**
     * The Vector that contains Interfaces that have been instantiated on
     * this router.  Default size = 4.
     */
    private Vector interfaces = new Vector(4);

    /**
     * The Vector of IDs for each interface that has been instantiated on
     * this router.  Default size = 4.
     */
    private Vector interfaceIDs = new Vector(4);

    /**
     * The Vector of talkers that have passed the registration process and
     * are authorized to talk on channels.
     */
    private Vector activeTalkers = new Vector();

    /**
     * The Hashtable of ResidentAgents that have been instantiated by the
     * ControlExecutive.
     */
    private Hashtable agents = new Hashtable();

    /**
     * The Hashtable of ResidentAgentCustomers that have registered to
receive
     * ResidentAgent replacements as they arrive.
     */
    private Hashtable agentCustomers = new Hashtable();

    /**
     * The Hashtable of MessageProcessors that have registered with this
     * ControlExecutive.
     */
    private Hashtable messageProcessors =
      new Hashtable();

    /**
     * The Hashtable of Channels that have been instantiated by this
ControlExecutive.
     */
    private Hashtable activeChannels = new Hashtable();

    /**
     * The Hashtable of channels that a given SaamTalker is registered to
talk on.
     */
    private Hashtable channelsTalkerHas = new Hashtable();

    /**
     * The Hashtable of channels that a given SaamListener is registered
to listen on.
     */
```

```
  private Hashtable channelsListenerHas = new Hashtable();

  /**
   * The Hashtable of Objects that have requested flows.
   */
  private Hashtable flowRequestors = new Hashtable();

  /**
   * The Hashtable of Objects that have been assigned flows.
   */
  private Hashtable assignedFlows = new Hashtable();


  /**
   * Instantiates and sets up communication with all Objects that are
necessary
   * to allow the ControlExecutive to start receiving ResidentAgents and
Messages.
   */
  public ControlExecutive(){
    mainGui = new MainGui(this,"SAAM Router Prototype");
    gui = new SAAMRouterGui(toString());

    transportInterface =
      new TransportInterface(this);

    //for receiving inbound packets
    packetFactory = new PacketFactory(this);

    arpCache = new ARPCache();

    //this should eventually become a ResidentAgent
    routingAlgorithm =
      new RoutingAlgorithm(this, arpCache);

    //Here is where the ControlExecutive registers itself as a
MessageProcessor
    registerMessageProcessor(this);

    /* Enable Talking on the channel that the Translator is listening on
for
       router status updates.*/
    try{
      addTalkerToChannel(this, ROUTER_STATUS_CHANNEL);
    }catch(ChannelException ce){
      gui.sendText(ce.toString());
    }

    try{
      //Get ownership of the SAAM_CONTROL_PORT so other applications
      //cannot.  When the TransportInterface sees a packet destined
      //for this port, it will not forward the packet directly on
      //the SAAM_CONTROL_PORT, rather, it will forward the packet
      //to the PacketFactory on the
ProtocolStackEvent.PACKETFACTORY_CHANNEL
      monitorPort(this, SAAM_CONTROL_PORT);
      gui.setTextField("Monitoring emulated port "+SAAM_CONTROL_PORT);
    }catch(PortAccessDeniedException pade){
      gui.sendText(pade.toString());
    }
    mainGui.updateDisplay();
```

```
}//ControlExecutive()

/**
 * Returns the IPv6Address of the server controlling this router
 * @return The IPv6Address of the server controlling this router.
 */
public IPv6Address getServerIP(){
  return serverIP;
}
/**
 * Returns the status of the ARPCache.
 * @return The status of the ARPCache ResidentAgent.  (if the ARPCache
 * contains an entry that corresponds to the next hop for flow zero,
this
 * method returns true).
 */
public boolean getArpCacheStatus(){
  return arpCacheReady;
}

/**
 * Returns the status of the EmulationTable.
 * @return The status of the EmulationTable.  (if the EmulationTable
 * contains an entry that corresponds to the next hop for flow zero,
this
 * method returns true).
 */
public boolean getEmulationTableStatus(){
  return emulationTableReady;
}

/**
 * There are three tables in every SAAM router:  The ARPCache, the
EmulationTable,
 * and the FlowRoutingTable.  Each of these tables notifies the
ControlExecutive
 * when it is ready to serve the router.  When all of these tables are
ready and
 * a few other conditions are met, the ControlExecutive sends a
notification to
 * the Translator.
 * @param o The Object sending the update.
 * @param status The status of o.
 */
public void updateCoreServiceStatus(
  Object o, boolean status){

  String className = o.getClass().getName();
  if(className.equals(
    "saam.residentagent.router.ARPCache")){
      arpCacheReady = status;
      updateRouterStatus();
  }else if (className.equals("saam.Translator")){
      emulationTableReady = status;
      updateRouterStatus();
  }
  //do nothing if another Object called this method
}

/**
```

```
    * The FlowRoutingTable uses this method to notify the
 ControlExecutive when it
    * is ready to serve the router.
    * @param o The Object sending the update.
    * @param flowZeroNextHop The IPv6Address of the next hop associated
 with flow zero.
    */
   public void updateCoreServiceStatus(
     Object o, IPv6Address flowZeroNextHop){

       String className = o.getClass().getName();
       this.flowZeroNextHop=flowZeroNextHop;
       boolean status = (flowZeroNextHop.equals(
           new IPv6Address()))? false:true;
       flowRoutingTableReady = status;
       //the following logic does not work for some reason...
       //I think it's the ArpCache.query method
       if(!arpCacheReady){
         if(routingAlgorithm.checkARPCache(
           new ARPCacheEntry(flowZeroNextHop))){
           arpCacheReady=true;
         }
       }
       updateRouterStatus();
   }

   /**
    * Returns the IPv6Address representing the next hop associated with
 flow zero.
    * @return The IPv6Address representing the next hop associated with
 flow zero.
    */
   public IPv6Address getFlowZeroNextHop(){
     return flowZeroNextHop;
   }

   /**
    * Performs a series of checks to determine the status of the router
 and
    * then updates the status accordingly.
    */
   private synchronized void updateRouterStatus(){
//     displayRouterStatus();
     String myAddress = null;
     try{
       myAddress = InetAddress.getLocalHost().getHostAddress();
     }catch(UnknownHostException uhe){}

     //compare local address with the address of the server.  If the
     //two addresses are the same and there is at least one interface
     //to process traffic, then the outbound interface for this server is
 ready.
     if(myAddress.equals(serverID.getIPv4()) &&
       (interfaces.size()>=1)){
       outboundInterfaceReady=true;
     }else if
       //otherwise, compare the network portion of the IPv6Address of the
 next
       //hop associated with flow zero to the network portions of the
 IPv6Addresses
```

```
        //of the interfaces on this router.  If there is a match, the
outbound
        //interface is ready.
      (routingAlgorithm.determineOutboundInterface(interfaces,
        flowZeroNextHop)!=null){
        outboundInterfaceReady=true;
      }

      //if all the conditions are met, notify the Translator that the
router
      //is ready.
      if(helloMessageReceived && arpCacheReady &&
       flowRoutingTableReady && emulationTableReady &&
       outboundInterfaceReady){

        if(routerStatus==ROUTER_DOWN){
          //send Hello Message to server
              routerStatus=ROUTER_UP;
          gui.sendText("\nThe Router is UP!  Sending Hello...");

          //Construct a new Hello Message to send to the Server
          Hello hello = new Hello(interfaceIDs);

          short sourcePort = (short)SAAM_CONTROL_PORT;
          RouterStatusEvent event = new RouterStatusEvent(
              toString(),
            this,
            ROUTER_STATUS_CHANNEL,
            routerStatus
          );
          //notify the Translator
          try{
              talk(event);
          }catch(ChannelException ce){
              gui.sendText(ce.toString());
          }
          try{
            int flowID = 0;
            gui.sendText("FlowID: "+flowID);
            short destPort = (short)SAAM_CONTROL_PORT;

            //send the Hello Message to the server.
//            try{
              send(this, hello, flowID,
                sourcePort,serverIP,destPort);
              gui.sendText("Hello sent:"+
                "flowID: "+flowID+
                "sourcePort: "+sourcePort+
                "serverIP: "+serverIP.toString()+
                "destPort: "+destPort);
            }catch(FlowException fe){
              gui.sendText(fe.toString());
            }
//          }catch(UnknownHostException uhe){
//            gui.sendText(uhe.toString());
//          }
        }
      }else{
        routerStatus=ROUTER_DOWN;
        //bringRouterDown();
        gui.sendText("Router is still down...");
```

```
    }
  }

  /**
   * Displays the current status of the router.
   */
  private void displayRouterStatus(){
    gui.sendText("\nCurrent Router Status:");
    gui.sendText(" helloMessageReceived:    "+helloMessageReceived);
    gui.sendText(" arpCacheReady:           "+arpCacheReady);
    gui.sendText(" flowRoutingTableReady:   "+
      flowRoutingTableReady);
    gui.sendText(" emulationTableReady:     "+emulationTableReady);
    gui.sendText(" outboundInterfaceReady: "+
      outboundInterfaceReady+
      "\n");
  }

  /**
   * In the SAAM architecture, traffic cannot be sent between hosts if
the
   * hosts are not assigned flows.  This method provides the mechanism
by
   * which hosts request flows from the SAAM server.  With the
information
   * provided in the parameters, this method constructs a
saam.message.FlowRequest.
   * It then sends that FlowRequest to the protocol stack for
transmission to
   * the SAAM server.  Note: If the requestor is not listening to a
port, the
   * requestor should first call the listenToRandomPort method for a
port assignment.
   * @param requestor The Object requesting the flow.
   * @param sourcePort The local port that requestor is listening on.
   * @param destHost The IPv6Address of the destination.
   * @param requestedDelay The amount of delay the requestor will
tolerate.
   * @param requestedLossRate The rate of loss the requestor will
tolerate.
   * @param requestedThroughput The amount of throughput the requestor
will tolerate.
   */
  public synchronized long requestFlow(ResidentAgent requestor,
   short sourcePort, IPv6Address destHost, int requestedDelay,
   int requestedLossRate, int requestedThroughput)
     throws FlowException{

      long timeStamp = System.currentTimeMillis();
      flowRequestors.put(new Long(timeStamp),requestor);
      IPv6Address sourceHost =
        ((InterfaceID)interfaceIDs.get(0)).getIPv6();
      FlowRequest request = new FlowRequest(
       sourceHost,
       destHost,
       timeStamp,
       requestedDelay,
       requestedLossRate,
       requestedThroughput);

      //FlowRequests travel on flow zero.
```

```
        int flowID = 0;
        short destPort = (short)SAAM_CONTROL_PORT;
        try{
          send(this, request, flowID, sourcePort,serverIP,destPort);
        }catch(FlowException fe){
          gui.sendText(fe.toString());
        }
        return timeStamp;
    }

    /**
     * Objects that have been assigned flows can send Messages with this
method.
     * In order to use this method, Objects must first request a flow
using the
     * requestFlow method; and then receive a flow assignment from the
server
     * @param sender The Object sending the message.
     * @param message The subclass of saam.message.Message to be sent.
     * @param flowID The flow ID that has been assigned for traffic from
sender
     *         destined for destHost.
     * @param sourcePort The port on the local machine that sender is
listening on.
     * @param destHost The IPv6Address of the destination.
     * @param destPort The port to which destHost is listening.
     */
    public synchronized void send(Object sender, Message message,
      int flowID, short sourcePort, IPv6Address destHost,
      short destPort) throws FlowException{

        if(sender.getClass().getName().equals(
          "saam.residentagent.router.LsaGenerator")){
            sender = this;
        }
        gui.sendText("Sending Message...");
        PacketFactory packetFactory = new PacketFactory();
        packetFactory.append(message);
        SAAMPacket saamPacket = null;
        try{
          saamPacket = new SAAMPacket(
            packetFactory.getBytes());
        }catch(UnknownHostException uhe){
          throw new FlowException(sender+
            " Problem building packet "+flowID);
        }
        //call the send method that takes an IPv6Packet,
        //using the IPv6Packet constructed by the TransportInterface
        IPv6Packet v6Packet = transportInterface.buildIPv6Packet(sender,
          saamPacket,flowID,sourcePort,destHost,destPort);
        gui.sendText(">> Message payload size = " +
v6Packet.getPayload().length);
        try{
          saamPacket = new SAAMPacket(v6Packet.getPayload());
        }catch(UnknownHostException uhe){
          throw new FlowException(sender+
            " Problem building packet "+flowID);
        }
        send(sender,v6Packet);
    }
```

```
  /**
   * Objects that have been assigned flows can send SAAMPackets with
this method.
   * In order to use this method, Objects must first request a flow
using the
   * requestFlow method; and then receive a flow assignment from the
server
   * @param sender The Object sending the message.
   * @param saamPacket The SAAMPacket to be sent.
   * @param flowID The flow ID that has been assigned for traffic from
sender
   *        destined for destHost.
   * @param sourcePort The port on the local machine that sender is
listening on.
   * @param destHost The IPv6Address of the destination.
   * @param destPort The port to which destHost is listening.
   */
  public synchronized void send(Object sender, SAAMPacket saamPacket,
    int flowID, short sourcePort, IPv6Address destHost,
    short destPort) throws FlowException{

      //call the send method that takes an IPv6Packet,
      //using the IPv6Packet constructed by the TransportInterface
      gui.sendText("Sending SAAM Packet...");
      IPv6Packet v6Packet = transportInterface.buildIPv6Packet(sender,
        saamPacket,flowID,sourcePort,destHost,destPort);
      send(sender,v6Packet);
  }//send()

  /**
   * Objects that have been assigned flows can send IPv6Packets with
this method.
   * In order to use this method, Objects must first request a flow
using the
   * requestFlow method; and then receive a flow assignment from the
server.
   * note: If the packet is destined for an Interface that is one this
router,
   * the packet will be delivered without flow id verification.
   * @param sender The Object sending the message.
   * @param ipv6Packet The IPv6Packet to be sent.
   */
  public synchronized void send(Object sender, IPv6Packet ipv6Packet)
    throws FlowException{

    int flowID = ipv6Packet.getHeader().getFlowLabel();
    //verify that the sender owns the flowID
    gui.sendText("Sending IPv6 Packet on flow "+flowID);
    if(!routingAlgorithm.isApplicationLayerPacket(ipv6Packet)){
      ResidentAgent agent = (ResidentAgent)assignedFlows.get(
            new Integer(flowID));
      if(sender.getClass().getName().equals("saam.server.Server")
        || (agent!=null&&agent.equals(sender)) || sender.equals(this)){
        //Here we forward the packet to an inbound interface
        //so it will be processed just as if it were an inbound
        //packet.
        ProtocolStackEvent event = new ProtocolStackEvent(
          sender.toString(),
          this,
          ProtocolStackEvent.getFromNICToInterfaceChannel(
            nextInboundInterface),
```

```
          ipv6Packet.getBytes());
        try{
          gui.sendText("Enqueuing packet for transmission at channel" +
            event.getChannel_ID());
          gui.sendText("   >> nextInboundInterface = " +
nextInboundInterface);
          talk(event);
        }catch(ChannelException ce){
          gui.sendText(ce.toString());
        }
//        routingAlgorithm.routeInboundPacket(ipv6Packet.getBytes());
        nextInboundInterface++;
        if(nextInboundInterface>=interfaces.size()){
          nextInboundInterface=0;
        }
      }else {
        gui.sendText(sender+
          " doesn't own flow "+flowID);
        throw new FlowException(sender+
          " doesn't own flow "+flowID);
      }
    }else{
      //this packet is destined for an Interface on this router,
      //so we forward it to the TransportInterface for delivery
      //on the proper emulated UDP port.
      IPv6Header v6Header = ipv6Packet.getHeader();

if(v6Header.getSource().toString().equals(IPv6Address.DEFAULT_HOST)){

v6Header.setSource(((Interface)interfaces.get(0)).getID().getIPv6());
        ipv6Packet.setHeader(v6Header);
      }

      gui.sendText("received app. layer packet from application layer");
      gui.sendText("forwarding to TransportInterface");
      SAAMPacket saamPacket = null;
      try{
        saamPacket = new SAAMPacket(ipv6Packet.getPayload());
      }catch(UnknownHostException uhe){
        gui.sendText(uhe.toString());
      }
      ProtocolStackEvent event = new ProtocolStackEvent(
        sender.toString(),
        //here we trick the TransportInterface into thinking this
        //event came from the routingAlgorithm.  Also, since the
        //routingAlgorithm is already registered to talk on this
        //channel, we make it past the security check that ensures
        //the talker is registered on the channel.
        routingAlgorithm,
        ProtocolStackEvent.
          FROM_ROUTINGALGORITHM_TO_TRANSPORTINTERFACE_CHANNEL,
        ipv6Packet.getBytes());
      try{
        talk(event);
      }catch(ChannelException tde){
        gui.sendText(tde.toString());
      }

    }
  }//send()
```

```
  /**
   * In order to send a flowRequest or any other type of traffic in a
SAAM network,
   * sending Objects must be listening to a port.  This method assigns
Objects a
   * random port that is higher than the highest well-known port, but no
higher than
   * MAX_PORT.
   * @param listener The SaamListener requesting a random port.
   */
  public int listenToRandomPort(SaamListener listener){
    int port = listenToRandomChannel(
      listener, HIGHEST_WELL_KNOWN_PORT+1,MAX_PORT);
    try{
      //the TransportInterface must be able to talk on the new port in
order
      //to deliver traffic to the listener
      addTalkerToChannel(transportInterface, port);
    }catch(ChannelException ce){
      gui.sendText(ce.toString());
    }
    return port;
      //also register the TransportInterface as a talker on
      //this port.
  }//listenToRandomPort()

  /**
   * Ports reside on Channels 0-MAX_PORT; any Channel higher than
MAX_PORT
   * is a Channel that is not associated with a port.  Examples of this
are
   * the communications Channels within the protocol stack.
   * @param listener The SaamListener requesting a random channel.
   * Note: Since this method is private, only the ControlExecutive can
   * assign listeners to a random channel.
   * @param lowestChannel The lowest channel in the range to be selected
from.
   * @param highestChannel The highest channel in the range to be
selected from.
   */
  private int listenToRandomChannel(
    SaamListener listener, int lowestChannel,
      int highestChannel){
    int channelFound = 0;
    boolean exception = true;
    while(exception){
      try{
        channelFound = (lowestChannel+(
          new Random()).nextInt(highestChannel-lowestChannel));
        addListenerToChannel(listener, channelFound);
        exception = false;
      }catch(Exception e){}
    }//while()
    return channelFound;
  }//listenToRandomChannel()

  /**
   * Returns the array of Strings representing the class
   * names of the messages this Object will register to process.
   * @return The array of Strings representing the class names
   * of the messages this Object will register to process.
```

```
    */
   public synchronized String[] getMessageTypes(){
     return messageTypes;
   }//getMessageTypes()

   /**
    * This private method is used by the ControlExecutive to perform the
    * steps necessary to instantiate an Interface.
    * @param id The InterfaceID that will be assigned to the interface
being
    * instantiated.  If an Interface with this id is already up, the
request
    * will be ignored.
    */
   private void standUpInterface(InterfaceID id){
     boolean alreadyActive = false;
     for (int i=0;i<interfaceIDs.size();i++){
       if(((InterfaceID)interfaceIDs.get(i)).equals(id)){
         alreadyActive = true;
         break;
       }
     }

     if(!alreadyActive){
       interfaceIDs.add(id);

       gui.sendText(" Instantiating interface["+
         interfaceCount++ +"]");
       gui.sendText(" IPv6Address: "+
         id.getIPv6().toString());
       Interface thisInterface =
         new Interface(this, id);
       interfaces.add(thisInterface);
       updateRouterStatus();
       routingAlgorithm.addInterface(thisInterface);
       try{
         addTalkerToChannel(this,
           ProtocolStackEvent.getFromNICToInterfaceChannel(
             interfaces.size()-1));
       }catch(ChannelException ce){
         gui.sendText(ce.toString());
       }
     }else{
       gui.sendText("Interface already active...");
     }
   }//standUpInterface()

   /**
    * This method contains the logic needed by the ControlExecutive
    * to process the Messages it is registered to process.
    * @param message The subclass of saam.message.Message to be
processed.
    */
   public void processMessage(Message message){
     String name = message.getClass().getName();
     if(name.equals(messageTypes[0])){
       InterfaceID id = (InterfaceID)message;
       standUpInterface(id);
       updateRouterStatus();
     }else if(name.equals(messageTypes[1])){
       try{
```

```
          serverID = ((ServerID)message);
          serverIP = serverID.getIPv6();
        }catch(Exception e){
          gui.sendText("Error processing serverID: "+e.toString());
        }
      }else if(name.equals(messageTypes[2])){
        gui.sendText("Got a FlowResponse.. ");
        FlowResponse response = (FlowResponse)message;
        long timeStamp = response.getTimeStamp();
        gui.sendText("TimeStamp: "+timeStamp);
        int flowID = response.getFlowId();
        gui.sendText("flowID: "+flowID);
        ResidentAgent requestor =
          (ResidentAgent)flowRequestors.get(new Long(timeStamp));
        gui.sendText("Forwarding to Requestor: "+requestor);
        if(requestor!=null){
          assignedFlows.put(new Integer(flowID),requestor);
          /*try{
            Thread.sleep(1000);
          }catch(InterruptedException ie){
            gui.sendText("ControlExecutive: "+ie.toString());
          }
          */
          requestor.receiveFlowResponse(response);
        }//else do nothing
      }else if(name.equals(messageTypes[3])){
        //received a saam.message.DemoHello from the DemoStation
        gui.sendText("received message type: " + messageTypes[3]);
        //
        Vector helloInterfaces = ((DemoHello)message).getInterfaceIDs();
        for(int i=0;i<helloInterfaces.size();i++){
          InterfaceID id = (InterfaceID)helloInterfaces.get(i);
          standUpInterface(id);
        }
        helloMessageReceived = true;
        updateRouterStatus();
        mainGui.updateDisplay();
      }
    }//processMessage()

    /**
     * Returns the number of Interfaces that have been stood up by this
ControlExecutive.
     * @return The number of Interfaces that have been stood up by this
ControlExecutive.
     */
    public int getNumberOfInterfaces(){
      return interfaceCount;
    }

    /**
     * This is the method MessageProcessors use to register to process
Messages.
     * The ControlExecutive retrieves the list of Messages from mp by
calling
     * mp.getMessageTypes().
     * @param mp The MessageProcessor that is registering with this
ControlExecutive.
     */
    public void registerMessageProcessor(MessageProcessor mp){
        gui.sendText("Registering MessageProcessor: "+mp);
```

```
      Object[] elementsIProcess = null;
      //retrieve the list of Messages
      elementsIProcess = mp.getMessageTypes();
      for(int i=0;i<elementsIProcess.length;i++){
        String element = (String)elementsIProcess[i];
        if(!element.equals("saam.message.FlowResponse")){
          MessageProcessor oldProcessor =
             (MessageProcessor)messageProcessors.put(element,mp);
         // notify old Processor that it will no longer
         // receive this type of message.
        }else{
          if(mp.equals(this)){
            messageProcessors.put(element,this);
          }else{
            gui.sendText("DENIED: "+element);
          }
        }
      }
  }

  /**
   * ResidentAgentCustomers use this method to register to receive
ResidentAgent
   * updates from the ControlExecutive when they arrive.  If an agent is
replaced
   * with a new agent, the ControlExecutive will call the customer's
replaceAgent
   * method.
   * @param rac the ResidentAgentCustomer requesting registration.
   */
  public void registerCustomer(ResidentAgentCustomer rac){
      Object[] agentsIUse = null;
      agentsIUse = rac.getAgentTypes();
      for(int i=0;i<agentsIUse.length;i++){
        String agent = (String)agentsIUse[i];
        Vector customers = null;
        synchronized(agentCustomers){
          customers = (Vector)agentCustomers.get(agent);
        }
        if(customers == null){
          customers = new Vector();
          agentCustomers.put(agent,customers);
        }
        customers.add(rac);
       // oldProcessor.
      }
  }

  /**
   * Replaces an existing ResidentAgent with an incoming ResidentAgent
   * of the same class name.  If there are multiple instances of the
existing
   * agent, the replacement will occur instance for instance.
   * @param classObject The Class of the incoming agent.
   * @param className The class name of the incoming ResidentAgent
subclass
   */
  private void replaceOldAgent(Class classObject, String className){

    boolean badAgent = false;
    Vector agentInstances = new Vector();
```

```
    int numberOfInstancesNeeded = 1;
    if(className.equals("saam.residentagent.router.Scheduler")){
      synchronized(interfaces){
        numberOfInstancesNeeded = interfaces.size();
      }
    }
    for (int i=0;i<numberOfInstancesNeeded;i++){
      try{
//          gui.sendText("About to install new agent");
        ResidentAgent newAgent = (ResidentAgent)
          classObject.newInstance();
        newAgent.install(this);
        gui.sendText("New agent installed");
        agentInstances.add(newAgent);
//          gui.sendText("Instances present: "+agentInstances.toString());
        numberOfSchedulersPresent++;
      }catch(Exception e){
        gui.sendText("ResidentAgent bad: "+e.toString());
        badAgent = true;
      }
    }
    if(!badAgent){
      gui.sendText("Agent "+
        (numberOfInstancesNeeded==0? "not instantiated.":"instantiated
"+
        (numberOfInstancesNeeded==1? "once.": numberOfInstancesNeeded+"
times")));

      boolean agentAlreadyInstalled = false;
      synchronized(agents){
        agentAlreadyInstalled=agents.containsKey(className);
      }
      if(agentAlreadyInstalled){
        gui.sendText(className+" already resident");
        Vector previousAgents = (Vector)agents.remove(className);
        gui.sendText("Uninstalling previous agent: ");
        gui.sendText("Removing from channels...");
        for(int i=0;i<previousAgents.size();i++){
          ResidentAgent previousAgent = (ResidentAgent)
            previousAgents.get(i);
          if(previousAgent instanceof SaamTalker){
            removeTalkerFromAllChannels(
                (SaamTalker)previousAgent);
          }
          //every ResidentAgent is a SaamListener
          removeListenerFromAllChannels(
              previousAgent);
          ResidentAgent replacement = (ResidentAgent)
            agentInstances.get(i);
          if(previousAgent!=null){
            gui.sendText("Previous agent uninstalling");
            previousAgent.transferState(replacement);
            previousAgent.uninstall();
            previousAgent = null;
          }else{
            gui.sendText("No previous agent installed");
          }
        }
      }
      for (int i=0;i<numberOfInstancesNeeded;i++){
        ResidentAgent replacement = (ResidentAgent)
```

```
          agentInstances.get(i);
        notifyAgentCustomers(replacement, className);
        gui.sendText("Notifying customers, agent: "+replacement);
      }
      agents.put(className,agentInstances);
    }
  }

  /**
   * Here, the ControlExecutive iterates through the Vector of
   * ResidentAgentCustomers and calls the replaceAgent method of
   * each customer, passing the new agent.
   */
  private void notifyAgentCustomers(
    ResidentAgent ra, String className){
      Vector customersOfThisAgent = (Vector)
        agentCustomers.get(className);
      if(customersOfThisAgent!=null){
        for(int i=0;i<customersOfThisAgent.size();i++){
          ((ResidentAgentCustomer)
            customersOfThisAgent.get(i)).
              replaceAgent(ra);
        }
        gui.sendText("Agent replaced: "+ra);
        gui.sendText("Customers: "+customersOfThisAgent);

      }
  }

  /**
   * In this method we would reflect into the Class Object and perform
   * a series of policy-related checks to determine whether or not the
   * agent is safe to instantiate.
   */
  private boolean examineAgent(Class classObject){
    //future work..
    return true;
  }

  /**
   * This method is called by the Channels this Object has registered to
   * monitor when a talker sends events on those Channels.
   * @param se The SaamEvent to be communicated.
   */
  public synchronized void receiveEvent(SaamEvent se){

    //the ControlExecutive only listens on the Channel between itself
and
    //the PacketFactory. Two types of traffic are sent on this Channel,
    //ResidentAgentEvents and MessageEvents

    if(se instanceof ResidentAgentEvent){
      Class classObject = ((ResidentAgentEvent)se).
        getClassObject();
      String className = classObject.getName();
      gui.sendText("received: "+className);
      if (examineAgent(classObject)){
        replaceOldAgent(classObject, className);
      }else{
        //notify someone that the agent failed the inspection
      }
```

```
    }else if (se instanceof MessageEvent){
      MessageEvent me = (MessageEvent)se;
      String name =
        me.getMessage().getClass().getName();
      gui.sendText("\nreceived: "+name);

      //call the appropriate MessageProcessor to handle this Message
      MessageProcessor mp = (MessageProcessor)
          messageProcessors.get(name);
      try{
        gui.sendText(" Calling Processor: "+mp.getClass().toString());
        mp.processMessage(me.getMessage());
      }catch(NullPointerException npe){
        //notify the sender that we do not have a
        //processor that is capable of processing this
        //Message.
        gui.sendText(" No Processor Available for " + name);
      }//try-catch
    }//not a ResidentAgentEvent or a MessageEvent
      mainGui.updateDisplay();
  }//receiveEvent()

  /**
   * Returns the Vector of Interfaces that have been instantiated by
this
   * ControlExecutive.
   * @return The Vector of Interfaces that have been instantiated by
this
   * ControlExecutive.
   */
  public Vector getInterfaces(){
      return interfaces;
  }//getInterfaces

  /**
   * The order in which Interfaces are instantiated is preserved.  Here
   * an Object can retrieve a specific Interface by instance number.
   * @param interfaceNumber The instance number of the Interface to be
   * retrieved.
   * @return The nth instance of Interface where n = interfaceNumber.
   */
  public Interface getInterface(int interfaceNumber){
      return (Interface)interfaces.get(interfaceNumber);
  }

  /**
   * Returns the Vector of InterfaceIDs assigned to the Interfaces
   * instantiated by this ControlExecutive.
   * @return The Vector of InterfaceIDs assigned to the Interfaces
   * instantiated by this ControlExecutive.
   */
  public Vector getInterfaceIDs(){
      return interfaceIDs;
  }//getInterfaces

  /**
   * Returns the Enumeration of Channels that have been instantiated
   * by this ControlExecutive.
   * @return The Enumeration of Channels that have been instantiated
   * by this ControlExecutive.
   */
```

```
  public Enumeration getActiveChannels(){
    return activeChannels.elements();
  }//getActiveChannels()

  /**
   * Returns true if the Channel has been instantiated by this
ControlExecutive.
   * @return True if the Channel has been instantiated by this
ControlExecutive.
   */
  public boolean isActiveChannel(int channel_ID){
    return activeChannels.containsKey(new Integer(channel_ID));
  }

  /**
   * To determine whether or not a talker is allowed to talk.  If
   * this method returns false, the talker will not be able to talk
   * on any Channels.
   * @param talker The SaamTalker to be verified.
   * @return True if the SaamTalker is allowed to talk.
   */
  private boolean verifyTalker(SaamTalker talker){
    return true;
  }//verifyRequestor()

  /**
   * To determine whether or not a listener has access to a given
Channel.
   * This method would be used to implement policy issues related to
access
   * control.
   * @param listener The listener to be verified.
   * @param channel_ID The ID of the Channel.
   */
  private boolean verifyChannelAccess(
    SaamListener pl, int channel_ID){
    //here, we would set the policy for channel_ID access.
    //i.e. we can restrict access of certain channel_ID to a
    //select list of listeners.  maybe a Hashtable called
    //"authorizationTable" which contains a Vector of
    //"authorizedListeners" and is keyed on channel_ID.
    return true;
  }//verifyAccess()

  /**
   * To determine whether or not a talker has access to a given Channel.
   * This method would be used to implement policy issues related to
access
   * control.
   * @param talker The talker to be verified.
   * @param channel_ID The ID of the Channel.
   */
  private boolean verifyChannelAccess(
    SaamTalker talker, int channel_ID){
    //here, we would set the policy for channel_ID access.
    //i.e. we can restrict access of certain channel_ID to a
    //select list of listeners.  maybe a Hashtable called
    //"authorizationTable" which contains a Vector of
    //"authorizedListeners" and is keyed on channel_ID.
    return true;
  }//verifyAccess()
```

```
/**
 * As the name implies, this method removes talker from the talker
 * Vectors of all Channels it has registered to talk on.
 * @param talker The talker to be removed.
 */
public void removeTalkerFromAllChannels(SaamTalker talker){
  if(channelsTalkerHas.containsKey(talker)){
    Vector channels = null;
    synchronized(channelsTalkerHas){
      channels =(Vector)channelsTalkerHas.get(talker);
    }
    Enumeration e = ((Vector)channelsTalkerHas.get(talker)).
      elements();
    while(e.hasMoreElements()){
      Channel thisChannel = (Channel)e.nextElement();
      thisChannel.removeTalker(talker);
      gui.sendText(talker.toString()+
        " removed from channel "+
        thisChannel.getChannel_ID());
      gui.sendText("The Vector: "+channels.toString());
    }
    channelsTalkerHas.remove(talker);
  }
}

/**
 * As the name implies, this method removes listener from the listener
 * Vectors of all Channels it has registered to listen on.
 * @param listener The listener to be removed.
 */
public void removeListenerFromAllChannels(
  SaamListener listener){

  if(channelsListenerHas.containsKey(listener)){
    Vector channels = null;
    synchronized(channelsListenerHas){
      channels = (Vector)channelsListenerHas.get(listener);
    }
    Enumeration e = channels.elements();
    while(e.hasMoreElements()){
      Channel thisChannel = (Channel)e.nextElement();
      thisChannel.removeListener(listener);
      gui.sendText(listener.toString()+
        " removed from channel "+
        thisChannel.getChannel_ID());
      gui.sendText("The Vector: "+channels.toString());
    }
    channelsListenerHas.remove(listener);
  }
}

/**
 * SaamTalkers use this method to attach themselves to a Channel.  If
this
 * method succeeds, the talker will be allowed to transmit events on
this
 * Channel.
 * @param talker The talker requesting permission to talk on a
Channel.
 * @param channel_ID The ID of the channel to be utilized.
```

```
     */
    public void addTalkerToChannel(SaamTalker talker, int channel_ID)
      throws ChannelException {

        if(!verifyTalker(talker)){
          throw new ChannelException("Talking Denied");
        }

        //now test to see whether this channel_ID is within the
        //range of channel_IDs on which this requestor is authorized
        //to talk (policy issue).
        if (!verifyChannelAccess(talker, channel_ID)){
            throw new ChannelException("Access Denied");
        }
        Channel channel = null;
        synchronized(activeChannels){
          channel = (Channel)activeChannels.get(new Integer(channel_ID));
        }
        if(channel==null) {
          channel = new Channel(channel_ID,talker);
        }
        activeChannels.put(new Integer(channel_ID),channel);
          channel.addTalker(talker);
          mainGui.updateDisplay();
        if(channelsTalkerHas.containsKey(talker)){
          synchronized(channelsTalkerHas){
            ((Vector)channelsTalkerHas.get(talker)).
              add(channel);
          }
        }else{
          Vector vectorOfChannels = new Vector();
          vectorOfChannels.add(channel);
          channelsTalkerHas.put(talker, vectorOfChannels);
        }
//        gui.sendText("Talker added:");
//        gui.sendText(channel.toString());
    }//addTalkerToChannel()

    /**
     * Allows a SaamListener to monitor an emulated UDP port
     * @param listener The listener requesting to monitor a port.
     * @param port The port to be monitored.
     */
   public void monitorPort(SaamListener listener, int port)
     throws PortAccessDeniedException{
     //presumably, the listener has already been verified
     //by the Control Executive and placed on an access
     //list within the EventController.  There is no such
     //access list at this time.
       try{
         if(!hasListener(port)){
          addTalkerToChannel(transportInterface,port);
          addListenerToChannel(listener, port);
//          gui.sendText(listener.toString()+
//              " listening to port: "+port);
         }else {
           gui.sendText(listener.toString()+
             " denied access to port: "+port);
           throw new PortAccessDeniedException("Port in use");
         }
       }catch(ChannelException ce){
```

```
        throw new PortAccessDeniedException("Not authorized");
      }//try-catch
  }

  /**
   * SaamListeners use this method to attach themselves to a Channel.
If this
   * method succeeds, the listener will receive all events that are sent
on this
   * Channel.
   * @param listener The listener requesting to monitor a Channel.
   * @param channel_ID The ID of the channel to be monitored.
   */
  public void addListenerToChannel(
    SaamListener listener, int channel_ID)
    throws ChannelException{

    if (verifyChannelAccess(listener,channel_ID)){

      Channel channel = null;
      synchronized(activeChannels){
        channel = (Channel)activeChannels.get(new Integer(channel_ID));
      }
      if(channel==null) {
        channel = new Channel(channel_ID,listener);
      }

      //no effect if the Channel is already on the active list
      activeChannels.put(new Integer(channel_ID),channel);
      channel.addListener(listener);
//      gui.sendText("Listener added:");
//      gui.sendText(channel.toString());
      if(channelsListenerHas.containsKey(listener)){
        synchronized(channelsListenerHas){
          ((Vector)channelsListenerHas.get(listener)).
            add(channel);
        }
      }else{
        Vector vectorOfChannels = new Vector();
        vectorOfChannels.add(channel);
        channelsListenerHas.put(listener, vectorOfChannels);
      }

    }//if
  }//addListenerToChannel()

  /**
   * Used to determine if any Objects are registered to listen on the
Channel
   * with channel_ID.
   * @param channel_ID The ID of the Channel to be queried.
   */
  public boolean hasListener(int channel_ID){
    try{
      Channel channel = null;
      synchronized(activeChannels){
        channel = (Channel)activeChannels.get(new Integer(channel_ID));
      }
      return channel.hasListeners();
    }catch(NullPointerException npe){}
      return (false);
```

```
  }//hasListener()

  /**
   * Used to determine if any Objects are registered to talk on the
Channel
   * with channel_ID.
   * @param channel_ID The ID of the Channel to be queried.
   */
  public boolean hasTalker(int channel_ID){
    try{
      Channel channel = null;
      synchronized(activeChannels){
        channel = (Channel)activeChannels.get(new Integer(channel_ID));
      }
      return channel.hasTalkers();
    }catch(NullPointerException npe){}
      return (false);

  }//hasListener()

  /**
   * Once approved to communicate on a channel, a
   * SaamTalker calls this method to actually broadcast
   * events on the channel.  A ChannelException will be
   * thrown if the talker is not registered to talk on
   * the channel contained in the SaamEvent.
   */
  public void talk(SaamEvent event) throws
    ChannelException{

      SaamTalker talker = event.getTalker();
      int channel_ID = event.getChannel_ID();
      Channel channel = null;
      synchronized(activeChannels){
// Questions to Dean:
// (1) Is the above sufficient?
// (2) Does this support talking to multiple channels?
// (3) Why are more and more ">>> Ready to ..." msgs printed out?
//
        channel = (Channel)
          activeChannels.get(new Integer(event.getChannel_ID()));
      }
      if(channel.isRegistered(talker)){
        //order the channel to notify its listeners
        gui.sendText("  >>> Ready to channel.talk");
        channel.talk(event);
        channel.setTimeLastUsed();
      }else{
        gui.sendText("ACCESS DENIED! Unregistered talker: "+
          talker.toString() +"\n"+
          "Attempted to talk on channel "+channel_ID);
        throw new ChannelException(
           talker.toString()+" not Registered on "+
           "channel "+channel_ID+".");
      }
  }//talk()

  /**
   * Displays the status of all Channels that have been instantiated by
the
```

```
   * ControlExecutive.  Channels are displayed in the ControlExecutive's
gui.
   * @param msg The text to appear before the channels are displayed.
   */
  public void displayActiveChannels(String msg){

/*      gui.sendText("\n"+msg);
      gui.sendText("Active channels:");
      Enumeration e = activeChannels.keys();
      while(e.hasMoreElements()){
        Integer key = (Integer)(e.nextElement());
        Channel channel =
          (Channel)activeChannels.get(key);
        gui.sendText(channel.toString());
      }//while(e.hasMoreElements())
*/
  }//displayActiveChannels()

  /**
   * Removes a SaamListener from the Vector of listeners associated with
the
   * Channel containing channel_ID
   * @param sl The SaamListener to be removed.
   * @param channel_ID The ID of the desired Channel.
   */
  public void removeListenerFromChannel(
    SaamListener sl, int channel_ID){

      Channel channel = null;
      synchronized(activeChannels){
         channel = (Channel)
          activeChannels.get(new Integer(channel_ID));
      }
      channel.removeListener(sl);
  }//closeChannelConnection()

  /**
   * Returns a <code>String</code> representation of this object
   * @return The <code>String</code> representation of this object
   */
  public String toString(){
    return ("Control Executive");
  }
}
```

```
package saam.control;

import java.util.Hashtable;
import java.io.File;

/**
 * A simple ClassLoader used by the ControlExecutive to load
 * ResidentAgents.
 */
class Loader extends ClassLoader {

  private Hashtable loadedClasses = new Hashtable();
  private Hashtable dynamicMessageTypes = new Hashtable();


  /**
  * Converts an array of bytes into an instance of class Class.
  * <p>
  *
  * @param name The expected name of the class, or null
  *  if not known, using '.' and not '/' as
  *  the separator and without a trailing ".class" suffix.
  * @param b the bytes that make up the class data. The bytes
  * should have the format of a valid class file.
  * @return The Class object that was created from b.
  */
  Class defClass (String name, byte[] b)
    throws LinkageError{
      Class theClass = null;
      try{
        theClass = defineClass(name, b, 0, b.length);
      }catch(LinkageError le){
        throw new LinkageError();
      }
      return theClass;
  }//defClass()

  /**
   * Stores byte arrays that presumably represent Java
   * class Objects.
   * @param name The String name of the class Object to be stored.
   * @param b The byte array that represents the class Object.
   */
  void putByteCode(String name, byte[] b){
    dynamicMessageTypes.put(name,b);
  }

  /**
   * Retrieves the byte array that represents the class Object
   * being retrieved by name.
   * @param name The String name of the class Object to be stored.
   * @return the byte array that represents the class Object.
   */
  byte[] getByteCode(String name){
    return (byte[])dynamicMessageTypes.get(name);
  }

  /**
   * Returns the Class Object that has been loaded.
   * @param name The String name of the class Object to be loaded.
   * @param resolve if true then resolve the class
```

```
   * @return The Class Object that has been loaded.
   * @throws ClassNotFoundException if the class could not be found.
   */
  public Class loadClass(String name, boolean resolve)
    throws ClassNotFoundException
  {
     try {
        Class newClass = (Class)loadedClasses.get(name);

        if (newClass == null) { // not yet loaded
           newClass = findSystemClass(name);
           if (newClass != null) return newClass;

           // class not found -- need to load it
           newClass = Class.forName(name);
           loadedClasses.put(name, newClass);
        }
        return newClass;
     } catch(ClassNotFoundException e) {
        throw new ClassNotFoundException(e.toString());
     }//if (newClass == null)
  }//loadClass()
}//Loader
```

```java
package saam.control;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Hashtable;
import java.util.TooManyListenersException;
import java.util.StringTokenizer;
import java.lang.reflect.Constructor;

import saam.net.*;
import saam.event.*;
import saam.message.*;
import saam.util.*;
import saam.residentagent.*;

/**
* A PacketFactory can be used to build SaamPackets for sending or
* to receive SaamPackets and extract their atomic elements.  These
* atomic elements are currently one of two types:  A subclass of
* saam.residentagent.ResidentAgent or a subclass of
* saam.message.Message.<p>
* A sender would instantiate a PacketFactory to build
* Saam Packets.  The PacketFactory's append methods receive
* Message Objects, ResidentAgent Objects, or a String that represents
* the class name of a ResidentAgent as parameters and then dynamically
* construct the appropriate header based on the number of elements
* received and the current time.  The getBytes method is used to
* retrieve the byte array that represents the SAAMPacket that has been
* constructed by this PacketFactory.<p>
* The ControlExecutive uses the PacketFactory to receive and parse
* SaamPackets.
*/
public class PacketFactory extends Thread
  implements SaamTalker, SaamListener{

  private final boolean guiActive = true;
  private SAAMRouterGui gui;
  private ControlExecutive controlExec;
  private boolean started = false;
  private boolean firstEvent = true;
  private boolean bytesRetrieved;
  private byte[] packet;
  private byte numberOfMessages;
  private Loader loader;
  private Class message;
  private SaamEvent currentEvent;
  private Thread owner;
  private static int instanceNumber;
  private Object theLock = new Object();

  /**
   * Use the no-args constructor to begin constructing packets
   * on the sending side.
   */
  //no-args constructor doesn't come for free when we have
  //another constructor
  public PacketFactory(){
    instanceNumber++;
    gui = new SAAMRouterGui(toString() + "("+
      instanceNumber+")");
```

```
    gui.setTextField("I construct outbound packets");
  }

  /**
   * This constructor is not available to Objects outside the
   * saam.control package.  The ControlExecutive uses this constructor
   * to receive and parse SAAMPackets.  The PacketFactory passes the
   * atomic elements (either ResidentAgents or Messages) up to the
   * ControlExecutive for further processing.
   * @param controlExec The ControlExecutive that is to receive
   * updates from this PacketFactory.
   */
  PacketFactory(ControlExecutive controlExec){
    this();
    gui.setTextField("I Listen for inbound packets");
    this.controlExec=controlExec;
    loader = new Loader();

    //******************************
    //**Listen to desired Channels**
    //******************************
    int channel_ID =
      ProtocolStackEvent.PACKETFACTORY_CHANNEL;
    try{
      controlExec.addListenerToChannel(this, channel_ID);
      gui.sendText("Listening to channel: "+channel_ID);
    }catch(ChannelException ce){
      gui.sendText(ce.toString());
    }//try-catch

    //****************************************
    //**Register to talk on desired Channels**
    //****************************************
    channel_ID = ControlExecutive.SAAM_CONTROL_PORT;
    try{
       controlExec.addTalkerToChannel(this,
         channel_ID);
       gui.sendText("Talking enabled on channel: " + channel_ID);
    }catch(ChannelException ce){
     gui.sendText(ce.toString());
    }
  }

  /**
   * When instantiated to receive packets, the PacketFactory
   * Thread waits until a SAAMPacket arrives, then it calls
   * the processPacket method.
   */
 public void run(){
    while(true){
        try{
          if(!started){
/*          synchronized(this){
              gui.sendText("Waiting...");
              while(!started) wait();
              started=true;
            }
*/
            synchronized(theLock){
              gui.sendText("Waiting...");
              while(!started) theLock.wait();
```

```
              started=true;
            }
          }
        }catch(InterruptedException ie){
          gui.sendText(ie.toString());
        }
        gui.sendText("Resumed");
        processPacket();
      }//while(started)
   }

   /**
    * This method is called by the Channels this Object has registered to
    * monitor when a talker sends events on those Channels.
    * @param se The SaamEvent to be communicated.

   public synchronized void receiveEvent(SaamEvent se){
  */
   public  void receiveEvent(SaamEvent se){

     gui.sendText("Got a packet");
     currentEvent=se;
     //check to see if the currentThread has an owner, if it
     //does, notify the owner that the event has arrived.
     //otherwise, just process the packet.
     if(!firstEvent){
       synchronized(theLock){
         theLock.notify();
       }
       if(!started){

         synchronized(theLock){
           started=true;
           theLock.notify();
         }

       }else{
         processPacket();
       }
     }else{
       firstEvent=false;
       started=true;
       start();
     }
     se=null;
   }
   /**
    * This method is used to extract the individual Class
    * Objects that are represented in the packet.  These Class
    * Objects are either of type 0 (ResidentAgent) or 1 (Message).<p>
    * If a ResidentAgent is received, a Class Object is created
    * that represents the agent.  That Class Object is then sent to
    * the ControlExecutive for screening and agent instantiation.<p>
    * If a Message is received, that Message is instantiated and sent
    * to the ControlExecutive for further processing.
    */
   private void processPacket() {
     int channel = currentEvent.getChannel_ID();
     String eventSource = (String)currentEvent.getSource();

     //packet is a byte array
```

```
packet = ((ProtocolStackEvent)currentEvent).getPacket();


//see saam.util for PrimitiveConversions and Array classes
long timeStamp = PrimitiveConversions.getLong(
    Array.getSubArray(packet,0,8));
numberOfMessages=packet[8];
gui.sendText("packet arrived: " +
  "\n  source:        " + eventSource +
  "\n  channel:       " + channel +
  "\n  size:          " + packet.length +
  "\n  # of updates:  " + numberOfMessages +
  "\n  timeStamp:     " + timeStamp);

//now we trim the packet by removing the header.
packet = Array.getSubArray(packet,9,packet.length);

//used to track the current position in the array.
int index = 0;

//extract and process each atomic element of the packet
//separately.  Here we assume the packet is a properly
//formatted SAAMPacket when it arrives, and that the
//length is less than the max allowed.
for(int i=1;i<=numberOfMessages;i++){
  gui.sendText("\nProcessing Element["+i+"]:");
  byte type = packet[index++];
  gui.sendText("  type:   "+type);

  //retrieve the number of bytes the class name occupies
  byte nameLength = packet[index++];

  //extract the name of the class file as a byte array
  byte[] elementNameArray = Array.getSubArray(
    packet,index, index+nameLength);
  index+=nameLength;

  //convert the name back into a String
  String elementName = new String(elementNameArray);
  gui.sendText("  Name: "+elementName);

  //retrieve the length of the Object
  short length = PrimitiveConversions.getShort(
    Array.getSubArray(packet,index,index+2));
  gui.sendText("  Length:       "+length);
  index+=2;

  //retrieve the bytecode of the Object
  byte[] bytes = Array.getSubArray(
    packet,index,index+length);
  index+=length;

  switch(type){
    case 0:
      gui.sendText("This is a ResidentAgent");
      //Assume this class is of type ResidentAgent
      try{
        //Attempt to define the class using the current
        //class loader.
        loader.defClass(elementName, bytes);
      }catch(LinkageError le){
```

```
      //If the loader already has a definition for the class
      //a LinkageError will be thrown.  If this happens, we
      //need to instantiate a new class loader and use it to
      //define the class.  A nice little trick we learned from
      //page 55 of Jason Hunter's "Java Servlet Programming" book.
      gui.sendText(le.toString());
      gui.sendText("Class was previously loaded...");
      gui.sendText("Replacing old ClassLoader...");
      Loader newLoader = new Loader();
      newLoader.defClass(elementName, bytes);
    }
    try{
      //message is of type Class.
      message = Class.forName(elementName, true, loader);
    }catch(ClassNotFoundException cnfe){
      gui.sendText(cnfe.toString());
    }
    gui.sendText(message.toString());
    ResidentAgentEvent rae = new ResidentAgentEvent(
      eventSource,
      this,
      ControlExecutive.SAAM_CONTROL_PORT,
      message);
    try{
      gui.sendText("Forwarding on channel "+
        ControlExecutive.SAAM_CONTROL_PORT);
      controlExec.talk(rae);
    }catch(ChannelException tde){
      gui.sendText(tde.toString());
    }

    break;
  case 1:
    gui.sendText("This is a Message");
    //Assume this class is of type Message.
    try{
      //message is of type Class.
      message = Class.forName(elementName);
    }catch(ClassNotFoundException cnfe){
      {gui.sendText("Bytecode for: "+elementName+
        " not found.");
      }
    }

    try{
      //Call the constructor from within this Class that
      //takes a byte array as its only argument
      Constructor cons = message.getConstructor(
        new Class[] {byte[].class});

      //Create the instance of this Message
      Message instance =
        (Message)cons.newInstance(
          new Object[] {bytes});
      gui.sendText(instance.toString());
      MessageEvent me = new MessageEvent(
        eventSource,
        this,
        ControlExecutive.SAAM_CONTROL_PORT,
        instance);
      //send this MessageEvent on the Control port.
```

```
          try{
            gui.sendText("Forwarding on channel "+
              ControlExecutive.SAAM_CONTROL_PORT);
            controlExec.talk(me);
          }catch(ChannelException tde){
            gui.sendText(tde.toString());
          }
        }catch(Exception e){
          //need to notify sender that we have no classfile
          //with this name
          gui.sendText(e.toString());
        }//try-catch
        break;
      default:
        gui.sendText("Packet type unrecognized: "+type);
        //packet type is unrecognized.  Here we could
        //extract a channel_ID that could be embedded
        //in the packet, and then send the unrecognized
        //element on that channel.
    }//switch
  }//for
  started=false;
}//processPacket()

/**
 * This method can be used to append a Message to an outgoing
 * SAAMPacket.  To later retrieve the entire packet (with header)
 * as a byte array, call the getBytes method.
 * @param me The Message to be appended.
 */
public void append(Message me){
  if(bytesRetrieved){
    packet=null;
    numberOfMessages=0;
    bytesRetrieved = false;
  }
  byte type = me.getType();
  String name = me.getClass().getName();
  byte nameLength = (byte)name.getBytes().length;
  byte[] parameters = me.getBytes();

  //here we could check the length of the parameter array supplied
  //with the length returned from the length() method call.
  short paramLength = (short)parameters.length;
  //now append the Message to the packet byte array
  packet = Array.concat(packet,type);
  packet = Array.concat(packet,nameLength);
  packet = Array.concat(packet,name.getBytes());
  packet = Array.concat(packet,
    PrimitiveConversions.getBytes(paramLength));
  packet = Array.concat(packet,parameters);
  //increment the count of messages in this packet
  numberOfMessages++;

  gui.sendText("Appended Message:" +
    "\n  Type:           " + type +
    "\n  name:           " + name +
    "\n  param length:   " + paramLength +
    "\n  # of elements:  " + numberOfMessages +
    "\n  packet length:  " + packet.length+"\n");
```

```java
    }

    /**
     * This method can be used to append a ResidentAgent to an outgoing
     * SAAMPacket.  To later retrieve the entire packet (with header)
     * as a byte array, call the getBytes method.
     * @param ra The ResidentAgent to be appended.
     */
    public void append(ResidentAgent ra) throws IOException{
      String name = ra.getClass().getName();
      append(name);
    }
    /**
     * This method can be used to append a ResidentAgent by name to an
     * outgoing SAAMPacket.  To later retrieve the entire packet
     * (with header) as a byte array, call the getBytes method.
     * @param residentAgentClassName The String name of the ResidentAgent
     * classfile to be appended.
     */
    public void append(String residentAgentClassName)
      throws IOException{
      if(bytesRetrieved){
        packet=null;
        numberOfMessages=0;
        bytesRetrieved = false;
      }
      byte type = 0;
      String name = residentAgentClassName;
      String fileName =
"C:\\WINNT\\Profiles\\administrator\\Desktop\\Java\\saamjuly\\saamxpand1
"+File.separatorChar +
//      String fileName = File.separatorChar +
        residentAgentClassName.replace('.',File.separatorChar);
      fileName+=".class";
      gui.sendText("File name: "+fileName);
      FileInputStream fis = null;
      try{
        fis = new FileInputStream(fileName);
      }catch(IOException ioe){
        throw new IOException(
          "Problem reading ResidentAgent: "+fileName);
      }
      byte nameLength = (byte)name.getBytes().length;
      byte[] byteCode = new byte[fis.available()];
      short length = (short)fis.read(byteCode);

      packet = Array.concat(packet,type);
      packet = Array.concat(packet,nameLength);
      packet = Array.concat(packet,name.getBytes());
      packet = Array.concat(packet,
        PrimitiveConversions.getBytes(length));
      packet = Array.concat(packet,byteCode);
      numberOfMessages++;

      gui.sendText("Appended ResidentAgent:" +
        "\n  Type:              " + type +
        "\n  name:              " + name +
        "\n  byteCode length:   " + length +
        "\n  # of elements:     " + numberOfMessages +
        "\n  packet length:     " + packet.length+"\n");
```

```
    }
    /**
     * Appends a header to the byte array.  The header conforms
     * to the structure of a SAAMHeader.
     */
    private void appendHeader(){
      byte[] timeStamp = PrimitiveConversions.getBytes(
          System.currentTimeMillis());
        packet = Array.concat(numberOfMessages,packet);
        packet = Array.concat(timeStamp,packet);
        gui.sendText("Appended header:"+
          "\n  timeStamp:    "+PrimitiveConversions.getLong(
            Array.getSubArray(packet,0,8))+
          "\n  # of updates: "+packet[8] +
          "\n  packet length: "+packet.length+"\n");

    }

    /**
     * Returns a byte array that conforms to the structure of
     * a SAAMPacket.
     * @return A byte array that conforms to the structure of
     * a SAAMPacket.
     */
    public byte[] getBytes(){
      appendHeader();
      bytesRetrieved = true;
      return packet;
    }

    /**
     * Returns the current length of the packet.
     * @return The current length of the packet.
     */
    public int length(){
      try{
        return packet.length;
      }catch(NullPointerException npe){
        return 0;
      }
    }

    /**
     * Returns a <code>String</code> representation of this object
     * @return The <code>String</code> representation of this object
     */
    public String toString(){
      return "Packet Factory";
    }

}
```

APPENDIX C. DEMO PACKAGE SOURCE CODE

```java
package saam.demo;

import java.net.*;
import java.io.IOException;
import java.util.Vector;

import saam.*;
import saam.control.*;
import saam.message.*;
import saam.residentagent.*;
import saam.router.*;
import saam.net.*;
import saam.util.*;

/**
 * This is a simple DemoStation used early on in testing the prototype.
 * It merely stands up two routers and a server.
 */
public class DemoStation {

   public static final int totalNumberOfInterfaces = 10;

   private PacketFactory packet = new PacketFactory();
   private Object[] IP;
   private byte[] macs = new byte[10];
   private Vector willowIDs = new Vector(4);
   private Vector peachIDs = new Vector(4);
   private Vector bonsaiIDs = new Vector(4);
   private Vector mapleIDs = new Vector(1);
   private Object[] nextHopV6;
   private InetAddress peachV4, willowV4, mapleV4, bonsaiV4, localV4;
   private byte type;
   private int destEmulationPort=9002;
   private SAAMRouterGui gui = new SAAMRouterGui("DemoStation");
   private IPv6Address v6Zero;
   private DatagramSocket socket;
   private ServerID serverID;
   private String[] coreAgents =
      {"saam.residentagent.router.Scheduler",
       "saam.residentagent.router.ARPCache",
       "saam.residentagent.router.FlowRoutingTable"};


   public static void main(String args[]){
      DemoStation test = new DemoStation();
   }

   public DemoStation(){
      try{
         socket = new DatagramSocket();
         gui.setTextField("My IP: "+
            InetAddress.getLocalHost().getHostAddress());
//          dest = InetAddress.getByName("127.0.0.1");

         //All of this constructor info could easily reside in
         //a database (or flat file, I suppose).

         //construct IPs that will reside on Willow's router
         //0
         IP = Array.concat(IP,
            IPv6Address.getByName(
```

```
        "99.99.99.0.0.0.0.0.0.0.0.0.0.0.0.0.2"));
//1
IP = Array.concat(IP,
   IPv6Address.getByName(
      "99.99.99.2.0.0.0.0.0.0.0.0.0.0.0.0.1"));
//2
IP = Array.concat(IP,
   IPv6Address.getByName(
      "99.99.99.1.0.0.0.0.0.0.0.0.0.0.0.0.1"));

//construct IPs that will reside on Peach's router
//3
IP = Array.concat(IP,
   IPv6Address.getByName(
      "99.99.99.2.0.0.0.0.0.0.0.0.0.0.0.0.2"));
//4
IP = Array.concat(IP,
   IPv6Address.getByName(
      "99.99.99.3.0.0.0.0.0.0.0.0.0.0.0.0.2"));
//5
IP = Array.concat(IP,
   IPv6Address.getByName(
      "99.99.99.5.0.0.0.0.0.0.0.0.0.0.0.0.1"));

//server IP
//6
IP = Array.concat(IP,
   IPv6Address.getByName(
      "99.99.99.0.0.0.0.0.0.0.0.0.0.0.0.0.1"));

//construct IPs that will reside on Bonsai's router
//7
IP = Array.concat(IP,
   IPv6Address.getByName(
      "99.99.99.1.0.0.0.0.0.0.0.0.0.0.0.0.2"));
//8
IP = Array.concat(IP,
   IPv6Address.getByName(
      "99.99.99.3.0.0.0.0.0.0.0.0.0.0.0.0.1"));
//9
IP = Array.concat(IP,
   IPv6Address.getByName(
      "99.99.99.4.0.0.0.0.0.0.0.0.0.0.0.0.1"));

//IPs of the interfaces on the router being instantiated
//on Willow
macs[0] = 2;
willowIDs.add(
  new InterfaceID((IPv6Address)IP[0],macs[0]));
macs[1] = 4;
willowIDs.add(
   new InterfaceID((IPv6Address)IP[1],macs[1]));

macs[2] = 3;
willowIDs.add(
   new InterfaceID((IPv6Address)IP[2],macs[2]));

//IP of the interface on the router being instantiated
//on Maple
macs[3] = 1;
mapleIDs.add(
```

```
                new InterfaceID((IPv6Address)IP[6],macs[3]));

            //IPs of the interfaces on the router being instantiated
            //on Peach
            macs[4] = 8;
            peachIDs.add(
              new InterfaceID((IPv6Address)IP[3],macs[4]));
            macs[5] = 9;
            peachIDs.add(
               new InterfaceID((IPv6Address)IP[4],macs[5]));

            macs[6] = 10;
            peachIDs.add(
               new InterfaceID((IPv6Address)IP[5],macs[6]));

            //IPs of the interfaces on the router being instantiated
            //on Bonsai
            macs[7] = 5;
            bonsaiIDs.add(
              new InterfaceID((IPv6Address)IP[7],macs[7]));
            macs[8] = 6;
            bonsaiIDs.add(
               new InterfaceID((IPv6Address)IP[8],macs[8]));

            macs[9] = 7;
            bonsaiIDs.add(
               new InterfaceID((IPv6Address)IP[9],macs[9]));

            //various IPs of nextHops
            for(int i=0;i<totalNumberOfInterfaces;i++){
              nextHopV6 =
                Array.concat(nextHopV6, (IPv6Address)IP[i]);
            }//for
            peachV4 = InetAddress.getByName("131.120.8.140");
            mapleV4 = InetAddress.getByName("131.120.8.142");
            willowV4 = InetAddress.getByName("131.120.8.133");
            bonsaiV4 = InetAddress.getByName("131.120.8.143");
            localV4 = InetAddress.getLocalHost();
            serverID = new ServerID((IPv6Address)IP[6],mapleV4);

        }catch(Exception uhe){
           gui.sendText(uhe.toString());
        }
/*        constructMaplePacket();
        try{
          Thread.sleep(7000);
        }catch(InterruptedException ie){}
*/
        constructWillowPacket();

        try{
          Thread.sleep(7000);
        }catch(InterruptedException ie){}
        sendWillowEmulationTableRemove();
/*
        constructPeachPacket();
*/
        socket.close();
    }

    /**
```

```
 * Constructs a SAAMPacket to stand up a router on the machine
 * named Willow.
 */
public void constructWillowPacket(){

   packet.append(serverID);

   //add Willow's three InterfaceIDs
   DemoHello helloMessage = new DemoHello(willowIDs);
   packet.append(helloMessage);

   try{
     //now append some ResidentAgents...
     //first the agents that are necessary for the
     //protocol stack
     for(int i=0;i<coreAgents.length;i++){
      gui.sendText("Appending core agent: "+coreAgents[i]);
      packet.append(coreAgents[i]);
     }
     //then any additional agents for the specific host
   }catch(IOException ioe){
      gui.sendText(ioe.toString());
   }

   //add one entry to the FlowRoutingTable
   int flowID = 0;
   byte sl = 0;
   packet.append(new FlowRoutingTableEntry(
     flowID,sl,(IPv6Address)nextHopV6[6]));

   packet.append(
      new EmulationTableEntry(
         (IPv6Address)nextHopV6[6],mapleV4));

   //this entry is erroneous and can be removed using
   //sendWillowEmulationTableRemove()
   packet.append(
      new EmulationTableEntry(
         (IPv6Address)nextHopV6[2],willowV4));

   packet.append(
      new EmulationTableEntry(
         (IPv6Address)nextHopV6[3],peachV4));

   packet.append(
      new EmulationTableEntry(
         (IPv6Address)nextHopV6[7],bonsaiV4));

   //add four entries to the ARPCache
   byte nextMAC = (byte)(1);
   packet.append(
     new ARPCacheEntry((IPv6Address)nextHopV6[6], nextMAC));

   nextMAC = (byte)(8);
   packet.append(
     new ARPCacheEntry((IPv6Address)nextHopV6[3], nextMAC));

   nextMAC = (byte)(5);
   packet.append(
     new ARPCacheEntry((IPv6Address)nextHopV6[7], nextMAC));
```

```java
    try{
       packet.append("saam.residentagent.router.LsaGenerator");
       packet.append("saam.residentagent.router.TwoWayFlows");
    }catch(IOException ioe){
       gui.sendText(ioe.toString());
    }
    //this is just too darn easy!
    //now send the packet
    byte[] packetArray = packet.getBytes();
    gui.sendText("#of updates: "+packetArray[8]);
    InetAddress dest = willowV4;
    try{
       DatagramPacket outboundPacket =
            new DatagramPacket(packetArray,packet.length(),
                                   dest,destEmulationPort);
       socket.send(outboundPacket);
    }catch(Exception e){
      gui.sendText(e.toString());
    }
    gui.sendText("Packet sent to "+dest.getHostAddress());
 }
 public void sendWillowEmulationTableRemove(){
    packet.append(
       new EmulationTableEntry((IPv6Address)nextHopV6[2]));

    byte[] packetArray = packet.getBytes();
    gui.sendText("#of updates: "+packetArray[8]);
    InetAddress dest = willowV4;
    try{
       DatagramPacket outboundPacket =
            new DatagramPacket(packetArray,packet.length(),
                                   dest,destEmulationPort);
       socket.send(outboundPacket);
    }catch(Exception e){
      gui.sendText(e.toString());
    }
    gui.sendText("Packet sent to "+dest.getHostAddress());
 }

 /**
  * Constructs a SAAMPacket to stand up a router on the machine
  * named Peach.
  */
 public void constructPeachPacket(){

    packet.append(serverID);

    //add Peach's three InterfaceIDs
    DemoHello helloMessage = new DemoHello(peachIDs);
    packet.append(helloMessage);

    try{
      //now append some ResidentAgents...
      //first the agents that are necessary for the
      //protocol stack
      for(int i=0;i<coreAgents.length;i++){
        packet.append(coreAgents[i]);
      }
      //then any additional agents for the specific host
    }catch(IOException ioe){
       gui.sendText(ioe.toString());
```

```
    }

    //add one entry to the FlowRoutingTable
    int flowID = 0;
    byte sl = 0;
    packet.append(new FlowRoutingTableEntry(
      flowID,sl,(IPv6Address)nextHopV6[1]));

    packet.append(
       new EmulationTableEntry(
           (IPv6Address)nextHopV6[1],willowV4));

    packet.append(
       new EmulationTableEntry(
           (IPv6Address)nextHopV6[8],bonsaiV4));

    //add two entries to the ARPCache
    byte nextMAC = (byte)(4);
    packet.append(
      new ARPCacheEntry((IPv6Address)nextHopV6[1], nextMAC));

    nextMAC = (byte)(6);
    packet.append(
      new ARPCacheEntry((IPv6Address)nextHopV6[8], nextMAC));

    try{
       packet.append("saam.residentagent.router.LsaGenerator");
       packet.append("saam.residentagent.router.TwoWayFlows");
    }catch(IOException ioe){
       gui.sendText(ioe.toString());
    }
    //this is just too darn easy!
    //now send the packet
    byte[] packetArray = packet.getBytes();
    gui.sendText("#of updates: "+packetArray[8]);
    try{
       DatagramPacket outboundPacket =
             new DatagramPacket(packetArray,packet.length(),
                                peachV4,destEmulationPort);
       socket.send(outboundPacket);
    }catch(Exception e){
      gui.sendText(e.toString());
    }
    gui.sendText("Packet sent to "+peachV4.getHostAddress());
}//constructPeachPacket()

/**
 * Constructs a SAAMPacket to stand up a server on the machine
 * named Maple.
 */
public void constructMaplePacket(){

    packet.append(serverID);

    //add Maple's InterfaceID
    DemoHello helloMessage = new DemoHello(mapleIDs);
    packet.append(helloMessage);

    try{
      //now append some ResidentAgents...
      //first the agents that are necessary for the
```

```
      //protocol stack
      for(int i=0;i<coreAgents.length;i++){
       packet.append(coreAgents[i]);
      }
      //then any additional agents for the specific host
      packet.append("saam.residentagent.server.ServerAgent");
    }catch(IOException ioe){
       gui.sendText(ioe.toString());
    }

    //add one entry to the FlowRoutingTable
    int flowID = 0;
    byte sl = 0;
    packet.append(new FlowRoutingTableEntry(
         flowID,sl,(IPv6Address)nextHopV6[6]));

    //add one entry to the FlowRoutingTable
    flowID = 1;
    sl = 0;
    packet.append(new FlowRoutingTableEntry(
         flowID,sl,(IPv6Address)nextHopV6[0]));

    //add one entry to the EmulationTable
    packet.append(
       new EmulationTableEntry(
          (IPv6Address)nextHopV6[6],mapleV4));

    //add one entry to the EmulationTable
    packet.append(
       new EmulationTableEntry(
          (IPv6Address)nextHopV6[0],willowV4));

    //add one entry to the ARPCache
    byte nextMAC = (byte)(1);
    packet.append(
     new ARPCacheEntry((IPv6Address)nextHopV6[6], nextMAC));

    //add one entry to the ARPCache
    nextMAC = (byte)(2);
    packet.append(
     new ARPCacheEntry((IPv6Address)nextHopV6[0], nextMAC));

    //this is just too darn easy!
    //now send the packet
    byte[] packetArray = packet.getBytes();
    gui.sendText("#of updates: "+packetArray[8]);
    try{
       DatagramPacket outboundPacket =
            new DatagramPacket(packetArray,packet.length(),
                                mapleV4,destEmulationPort);
       socket.send(outboundPacket);
    }catch(Exception e){
      gui.sendText(e.toString());
    }
    gui.sendText("Packet sent to "+mapleV4.getHostAddress());

    //let's send another

  }
}
```

```
import java.net.*;
import java.io.IOException;

import saam.*;
import saam.net.*;
import saam.router.*;
import saam.util.*;
import saam.event.*;

public class SAAMPacketSender {

   private static final int packetCount=4;
   private PacketFactory packet = new PacketFactory();
   private byte[] packetArray;
   private InetAddress dest;
   private int destEmulationPort=9001;
   private SAAMRouterGui gui;

   public static void main(String args[]){
      SAAMPacketSender test = new SAAMPacketSender();
   }

   public SAAMPacketSender(){
      gui = new SAAMRouterGui("SAAMPacketSender");
      try{
         gui.setTextField("My IP: "+
            InetAddress.getLocalHost().getHostAddress());
         dest = InetAddress.getByName("131.120.1.87");
         //construct the IPv6Header
      }catch(UnknownHostException uhe){
         gui.sendText(uhe.toString());
      }
      constructPacket();
   }

   public void constructPacket(){
      //stuff all this into the packet
      packet = new PacketFactory();

      //construct and append the MAC
      byte MAC = (byte)4;
      packet.append(MAC);

      //construct and append the IPv6Header
      int flowLabel = 0;
      IPv6Address v6Source = null;
      IPv6Address v6Dest = null;
      try{
         v6Source = IPv6Address.getByName(
                  "99.99.99.2.0.0.0.0.0.0.0.0.0.0.0.2");
         v6Dest = IPv6Address.getByName(
                  "99.99.99.2.0.0.0.0.0.0.0.0.0.0.0.1");
      }catch(UnknownHostException uhe){
         gui.sendText(uhe.toString());
      }
      IPv6Header v6Header = new IPv6Header(
         flowLabel,v6Source, v6Dest);
      packet.append(v6Header.getHeader());

      //construct and append the UDPHeader
      short sourcePort = 8000;
```

```
short destPort = ApplicationEvent.SAAM_CONTROL_PORT;
short payloadLength = 5000;
short checksum = 0;
UDPHeader udpHeader = new UDPHeader(sourcePort,
   destPort,payloadLength,checksum);
packet.append(udpHeader.getHeader());


//next we construct and append a SAAMPacket that
//contains two FlowRoutingTableAdds and one
//FlowRoutingTableRemove.  This process
//involves first constructing the payload which
//will consist of the three updates, and
//then constructing the SAAMHeader, then calling the
//constructor to SAAMPacket, providing the SAAMHeader
//and payload as parameters.  Then we append the
//SAAMPacket to the outbound packet.
byte numberOfSaamUpdates=0;
//construct the payload for the SAAMPacket
PacketFactory saamPayload = new PacketFactory();

/////////////////////////////////////////////
//add two entries to the FlowRoutingTable
byte type = 4;
int[] flowID = {1024,1025};
byte[] sl = {3,3};
FlowRoutingTableAdd[] frtAdd = new FlowRoutingTableAdd[2];
for(int i=0;i<2;i++){
   //constructing a FlowRoutingTableAdd and calling the getBytes
   //method here is easier than manually converting the int
   //flowID to a 3-byte array.
   IPv6Address nextHop = null;
   try{
      nextHop = IPv6Address.getByName(
         "99.99.99.2.0.0.0.0.0.0.0.0.0.0.0.2");
   }catch(UnknownHostException uhe){
      gui.sendText(uhe.toString());
   }
   frtAdd[i] = new FlowRoutingTableAdd(
            flowID[i],sl[i],nextHop);
   saamPayload.append(type);
   saamPayload.append(frtAdd[i].getBytes());
   numberOfSaamUpdates++;
   try{
   gui.sendText((new FlowRoutingTableAdd(
      frtAdd[i].getBytes()).toString()));
   }catch(UnknownHostException uhe){}
}//for
//demonstrate how to remove an entry... here we remove
//the last entry made
type = 5;
saamPayload.append(type);
   FlowRoutingTableRemove frtRemove =
      new FlowRoutingTableRemove(flowID[0]);
   saamPayload.append(frtRemove.getBytes());
   numberOfSaamUpdates++;
   gui.sendText(frtRemove.toString());
/////////////////////////////////////////////

//construct the SAAMHeader
SAAMHeader saamHeader = new SAAMHeader(
```

```
            255,numberOfSaamUpdates);
        gui.sendText("Saam header:\n  "+saamHeader.toString());
        gui.sendText("numberOfSaamUpdates: "+numberOfSaamUpdates);

        //construct the SAAMPacket by providing the constructor
        //the SAAMHeader and saamPayload we just created.
        SAAMPacket saamPacket = new SAAMPacket(saamHeader,
            saamPayload.getBytes());

        //now append the saamPacket to the outbound packet
        packet.append(saamPacket.getBytes());

        //convert the entire packet to a byte array and send it.
        packetArray = Array.concat(packetArray,packet.getBytes());
        //this is just too darn easy!
        //now send the packet
        try{
            gui.sendText("packetArray.length: "+packetArray.length);
            DatagramPacket outboundPacket =
                 new DatagramPacket(packetArray,packetArray.length,
                                    dest,destEmulationPort);
            (new DatagramSocket()).send(outboundPacket);
        }catch(Exception e){
          gui.sendText(e.toString());
        }
        gui.sendText("Packet sent to "+dest.getHostAddress());
    }//constructPacket()
}
```

APPENDIX D. EVENT PACKAGE SOURCE CODE

```java
package saam.event;

import saam.net.SAAMPacket;

/**
 * ApplicationEvents are used by the Transport Interface to
 * notify port listeners that a packet has arrived that is
 * destined for a specific port.
 */
public class ApplicationEvent extends SaamEvent {

      /**
       * The byte array representing the packet.
   * @serial
       */
   private byte[] packet;

   /**
    * Constructs an ApplicationEvent with the associated
    * attributes.
    * @param eventSource The original source of the event.
    * @param talker The SaamTalker that sent the event.
    * @param port The port the packet is destined for.
    * @param packet The byte array representing the arriving packet.
    */
   public ApplicationEvent(String eventSource,
     SaamTalker talker, int port, byte[] packet){
          super(eventSource,talker,port);
          this.packet = packet;
   }

   /**
    * Returns the port associated with this ApplicationEvent.
    * @return The port associated with this ApplicationEvent.
    */
   public int getPort(){
     return super.getChannel_ID();
   }

  /**
      * Returns the entire packet as a byte array.
      * @return The entire packet as a byte array.
      */
   public byte[] getPacket(){
    return packet;
   }

}
```

```java
package saam.event;

/**
 * ChannelExceptions are thrown for various reasons.  For
 * example, if a SaamTalker is not authorized to talk on
 * a Channel, but requests to do so, this exception is thrown.
 */
public class ChannelException extends Exception{

  /**
   * Constructs a generic ChannelException
   */
  public ChannelException() {
    super();
  }

  /**
   * Constructs a ChannelException with a descriptive String.
   * @param s A String describing the Exception in more detail.
   */
  public ChannelException(String s) {
    super(s);
  }
}
```

```java
package saam.event;

import saam.message.Message;

/**
 * A MessageEvent is a simple SaamEvent that contains a Message
 * and a port on which the Message will be sent. The
 * PacketFactory constructs and sends MessageEvents to the
 * ControlExecutive when Messages are received.
 */
public class MessageEvent extends SaamEvent {

  /**
   * @serial
   */
   private Message message;


   /**
    * Constructs a MessageEvent with the attributes provided.
    * @param eventSource The location from which the Message originated.
    * @param talker The Object that instantiated the MessageEvent.
    * @param port The port on which the Message is to be passed.
    * @param message The Message to be passed.
    */
   public MessageEvent(String eventSource,
     SaamTalker talker, int port, Message message){
          super(eventSource,talker,port);
          this.message = message;
   }

   /**
    * Returns the UDP port associated with this MessageEvent.
    * @return The UDP port associated with this MessageEvent.
    */
   public int getPort(){
     return super.getChannel_ID();
   }

   /**
    * Returns the Message associated with this MessageEvent.
    * @return The Message associated with this MessageEvent.
    */
   public Message getMessage(){
    return message;
   }
}
```

```java
package saam.event;

/**
 * PortAccessDeniedException are thrown for various reasons.  For
 * example, if a ResidentAgent is not authorized to talk on
 * a particular port, but requests to do so, this exception is thrown.
 */
public class PortAccessDeniedException extends Exception{

  /**
   * Constructs a generic PortAccessDeniedException
   */
  public PortAccessDeniedException() {
    super();
  }

  /**
   * Constructs a PortAccessDeniedException with a descriptive String.
   * @param s A String describing the Exception in more detail.
   */
  public PortAccessDeniedException(String s) {
    super(s);
  }
}
```

```
/**
 * A container for all methods associated with event-handling
 * within the SAAM architecture.
 */
package saam.event;

import saam.net.IPv6Address;

/**
 * A ProtocolStackEvent is an event that is fired from within
 * the protocol stack.  Normally this class is used as a means
 * of communications between Objects within the protocol stack,
 * such as to pass a packet from the Interface to the RoutingAlgorithm.
 */
public class ProtocolStackEvent extends SaamEvent {

  /**
   * The ID of the Channel used primarily to pass packets from the
   * RoutingAlgorithm to the TransportInterface
   */
  public static final int
    FROM_ROUTINGALGORITHM_TO_TRANSPORTINTERFACE_CHANNEL
                                                   = 80001;

  /**
   * The ID of the Channel used primarily to initiate
LinkStateAdvertisements.
   */
  public static final int LSA_CHANNEL              = 80002;

  /**
   * The ID of the Channel used primarily to pass packets from the
   * Translator to all of the NetworkInterfaceCards simultaneously.
   */
  public static final int
    FROM_TRANSLATOR_TO_NICS_CHANNEL                = 80003;
  /**
   * The ID of the Channel used primarily to pass packets to the
PacketFactory.
   */
  public static final int
    PACKETFACTORY_CHANNEL                          = 80004;

  /**
   * The ID of the Channel used primarily to pass packets from any
   * NetworkInterfaceCard to the Translator.
   */
  public static final int
    FROM_NICS_TO_TRANSLATOR_CHANNEL                = 80005;

  /**
   * The ID of the Channel used primarily to pass packets from the
   * TransportInterface to the RoutingAlgorithm.
   */
  public static final int
    FROM_TRANSPORTINTERFACE_TO_ROUTINGALGORITHM_CHANNEL
                                                   = 80006;
  /**
   * The ID of the Channel used primarily to pass packets from the
   * first instantiated NetworkInterfaceCard to the first instantiated
   * Interface.
```

```
  */
 public static final int
   FROM_NIC_TO_INTERFACE_START_CHANNEL        = 80300;

 /**
  * The ID of the Channel used primarily to pass packets from the first
  * instantiated Interface and destined for the inbound Queue
associated
  * with that Interface.
  */
 public static final int
   ENQUEUING_INBOUND_PACKET_START_CHANNEL    = 80400;

 /**
  * The ID of the Channel used primarily to pass packets from the
  * RoutingAlgorithm to the first instantiated Interface.
  */
 public static final int
   FROM_ROUTINGALGORITHM_TO_INTERFACE_START_CHANNEL
                                             = 80500;
 /**
  * The ID of the Channel used primarily to pass packets from the
  * first instantiated Interface to a Service Level Queue of that
  * Interface
  */
 public static final int
   FROM_INTERFACE_TO_SLQUEUE_START_CHANNEL   = 80600;

 /**
  * The ID of the Channel used primarily to pass packets from a
  * Service Level Queue to the Scheduler associated with the Interface
  * to which that Queue belongs.
  */
 public static final int
   FROM_SLQUEUE_TO_SCHEDULER_START_CHANNEL   = 80700;


 /**
  * The ID of the Channel used primarily to pass packets from the
  * first instantiated Scheduler to the NetworkInterfaceCard associated
  * with that Scheduler.
  */
 public static final int
   FROM_SCHEDULER_TO_NIC_START_CHANNEL       = 80800;


     /**
      * The byte array representing the packet.
  * @serial
      */
  private byte[] packet;

  public static final int DEFAULT_SERVICE_LEVEL = 3;

 /**
  * Represents the service level associated with this event.
  * @serial
  */
  private int   serviceLevel = DEFAULT_SERVICE_LEVEL;

 /**
```

```
 * Represents the next hop associated with this event.
 * @serial
 */
 private IPv6Address nextHop;


/**
 * Constructs a ProtocolStackEvent with the given attributes.
 * @param eventSource The object where the event originated.
 * @param talker The SAAMTalker that instantiated this event.
 * @param channel_ID The ID of the channel this event occured on.
 * @param packet The byte array representing the packet.
 */
 public ProtocolStackEvent(String eventSource,
   SaamTalker talker, int channel_ID, byte[] packet){
        super(eventSource,talker,channel_ID);
        this.packet = packet;
 }

/**
 * Constructs a ProtocolStackEvent with the given attributes.
 * @param eventSource The object where the event originated.
 * @param talker The SAAMTalker that instantiated this event.
 * @param channel_ID The ID of the channel this event occured on.
 * @param packet The byte array representing the packet.
 * @param serviceLevel The service level associated with this event.
 */
 public ProtocolStackEvent(String eventSource,
   SaamTalker talker, int channel_ID, byte[] packet,
   int serviceLevel){
        super(eventSource, talker, channel_ID);
        this.packet = packet;
        this.serviceLevel = serviceLevel;
 }

/**
 * Constructs a ProtocolStackEvent with the given attributes.
 * @param eventSource The object where the event originated.
 * @param talker The SAAMTalker that instantiated this event.
 * @param channel_ID The ID of the channel this event occured on.
 * @param packet The byte array representing the packet.
 * @param serviceLevel The service level associated with this event.
 * @param IPv6Address nextHop The IPv6Address of the next hop.
 */
 public ProtocolStackEvent(String eventSource,
   SaamTalker talker, int channel_ID, byte[] packet,
   int serviceLevel, IPv6Address nextHop){
        super(eventSource, talker, channel_ID);
        this.packet = packet;
        this.serviceLevel = serviceLevel;
        this.nextHop = nextHop;
 }

 /**
  * Returns an int representing the Channel primarily used for traffic
  * originating at the NetworkInterfaceCard specified and destined for
  * the Interface with the same number as nicNumber.
  * @param nicNumber The instance number of the NetworkInterfaceCard
  * @return An int representing the Channel primarily used for traffic
  * originating at the NetworkInterfaceCard specified and destined for
  * the Interface with the same number as nicNumber.
```

```
 */
public static int getFromNICToInterfaceChannel(
  int nicNumber){
  return FROM_NIC_TO_INTERFACE_START_CHANNEL+
    nicNumber;
}
/**
 * Returns an int representing the Channel primarily used for traffic
 * originating at the RoutingAlgorithm and destined for
 * the Interface with interfaceNumber.
 * @param interfaceNumber The instance number of the Interface
 * @return An int representing the Channel primarily used for traffic
 * originating at the RoutingAlgorithm and destined for
 * the Interface with interfaceNumber.
 */
public static int getFromRoutingAlgorithmToInterfaceChannel(
  int interfaceNumber){
  return FROM_ROUTINGALGORITHM_TO_INTERFACE_START_CHANNEL+
    interfaceNumber;
}
/**
 * Returns an int representing the Channel primarily used for traffic
 * originating at the Interface with interfaceNumber and destined
 * for the inbound Queue associated with that Interface.
 * @param interfaceNumber The instance number of the Interface
 * @return An int representing the Channel primarily used for traffic
 * originating at the the Interface with interfaceNumber and destined
 * for the inbound Queue associated with that Interface.
 */
public static int getEnqueuingInboundPacketChannel(
  int interfaceNumber){
  return ENQUEUING_INBOUND_PACKET_START_CHANNEL+interfaceNumber;
}
/**
 * Returns an int representing the Channel primarily used for traffic
 * originating at the Interface with interfaceNumber and destined for
 * the service level Queue associated with that interface.
 * @param interfaceNumber The instance number of the Interface
 * @return An int representing the Channel primarily used for traffic
 * originating at the Interface with interfaceNumber and destined for
 * the service level Queue associated with that interface.
 */
public static int getFromInterfaceToSLQueueChannel(
  int interfaceNumber){
  return FROM_INTERFACE_TO_SLQUEUE_START_CHANNEL+
    interfaceNumber;
}
/**
 * Returns an int representing the Channel primarily used for traffic
 * originating at the Scheduler specified by interfaceNumber and
 * destined for the NetworkInterfaceCard with the same number as
 * interfaceNumber.
 * @param interfaceNumber The instance number of the Interface.
 * @return An int representing the Channel primarily used for traffic
 * originating at the Scheduler specified by interfaceNumber and
 * destined for the NetworkInterfaceCard with the same number as
 * interfaceNumber.
 */
public static int getFromSchedulerToNICChannel(
  int interfaceNumber){
  return FROM_SCHEDULER_TO_NIC_START_CHANNEL+
```

```
            interfaceNumber;
    }

    /**
     * Returns the entire packet as a byte array.
     * @return The entire packet as a byte array.
     */
    public byte[] getPacket(){
        return packet;
    }

    /**
     * Sets the byte array defined as the packet.
     * @param packet The new value for the packet.
     */
    public void setPacket(byte[] packet){
        this.packet = packet;
    }

    /**
     * Returns the service level associated with this event.
     * @return The service level associated with this event.
     */
    public int getServiceLevel(){
        return serviceLevel;
    }

    /**
     * Returns the next hop associated with this event.
     * @return The next hop associated with this event.
     */
    public IPv6Address getNextHop(){
        return nextHop;
    }
}
```

```
package saam.event;

import saam.residentagent.ResidentAgent;

/**
 * A ResidentAgentEvent is a simple SaamEvent that contains a
ResidentAgentEvent
 * and a port on which the ResidentAgentEvent will be sent. The
 * PacketFactory constructs and sends ResidentAgentEvents to the
 * ControlExecutive when ResidentAgents are received.
 */
public class ResidentAgentEvent extends SaamEvent {

  /**
   * @serial
   */
   private Class classObject;


   /**
    * Constructs a ResidentAgentEvent with the attributes provided.
    * @param eventSource The location from which the ResidentAgent
originated.
    * @param talker The Object that instantiated the ResidentAgentEvent.
    * @param port The port on which the ResidentAgent is to be passed.
    * @param classObject The Class Object representing the ResidentAgent
to be
    *  passed.
    */
   public ResidentAgentEvent(String eventSource,
     SaamTalker talker, int port, Class classObject){
          super(eventSource,talker,port);
          this.classObject = classObject;
   }

   /**
    * Returns the UDP port associated with this ResidentAgentEvent.
    * @return The UDP port associated with this ResidentAgentEvent.
    */
   public int getPort(){
     return super.getChannel_ID();
   }

   /**
    * Returns the Class Object associated with this ResidentAgentEvent.
    * @return The Class Object associated with this ResidentAgentEvent.
    */
   public Class getClassObject(){
    return classObject;
   }
}
```

```java
package saam.event;

import saam.residentagent.ResidentAgent;

/**
 * A ResidentAgentEvent is a simple SaamEvent that contains a
ResidentAgentEvent
 * and a port on which the ResidentAgentEvent will be sent. The
 * PacketFactory constructs and sends ResidentAgentEvents to the
 * ControlExecutive when ResidentAgents are received.
 */
public class ResidentAgentEvent extends SaamEvent {

   /**
    * @serial
    */
    private Class classObject;


    /**
     * Constructs a ResidentAgentEvent with the attributes provided.
     * @param eventSource The location from which the ResidentAgent
originated.
     * @param talker The Object that instantiated the ResidentAgentEvent.
     * @param port The port on which the ResidentAgent is to be passed.
     * @param classObject The Class Object representing the ResidentAgent
to be
     *  passed.
     */
    public ResidentAgentEvent(String eventSource,
      SaamTalker talker, int port, Class classObject){
           super(eventSource,talker,port);
           this.classObject = classObject;
    }

    /**
     * Returns the UDP port associated with this ResidentAgentEvent.
     * @return The UDP port associated with this ResidentAgentEvent.
     */
    public int getPort(){
      return super.getChannel_ID();
    }

    /**
     * Returns the Class Object associated with this ResidentAgentEvent.
     * @return The Class Object associated with this ResidentAgentEvent.
     */
    public Class getClassObject(){
     return classObject;
    }
}
```

```
package saam.event;

/**
 * A RouterStatusEvent is a simple SaamEvent that contains a boolean
 * and a port on which the RouterStatusEvent will be sent. The
 * ControlExecutive constructs and sends RouterStatusEvents to the
 * Translator when the status of the router changes.
 */
public class RouterStatusEvent extends SaamEvent {

      /**
       * The boolean representing the status of the router.
   * @serial
       */
   private boolean status;

   /**
    * Constructs a RouterStatusEvent with the attributes provided.
    * @param eventSource The location from which the RouterStatusEvent
originated.
    * @param talker The Object that instantiated the RouterStatusEvent.
    * @param port The port on which the RouterStatusEvent is to be
passed.
    * @param status The boolean representing the status of the router.
    */
   public RouterStatusEvent(String eventSource,
     SaamTalker talker, int port, boolean status){
          super(eventSource,talker,port);
          this.status = status;
   }

   /**
    * Returns the UDP port associated with this RouterStatusEvent.
    * @return The UDP port associated with this RouterStatusEvent.
    */
   public int getPort(){
     return super.getChannel_ID();
   }

  /**
      * Returns true if the router is up.
      * @return True if the router is up.
      */
   public boolean getStatus(){
    return status;
   }

}
```

```java
package saam.event;

import java.util.EventObject;

/**
 * This abstract class provides the structure for a basic
 * event within the Saam architecture.  Every SaamEvent has an event
source,
 * a channel_ID, and a SaamTalker.
 */
public abstract class SaamEvent extends EventObject {

  /**
   * @serial
   */
  private int channel_ID;

  /**
   * @serial
   */
  private SaamTalker talker;

  /**
  * Constructs a SaamEvent with the attributes provided.
  * @param eventSource The location from which the event originated.
  * @param talker The Object that instantiated the event.
  * @param channel_ID The ID of the Channel on which the event is
  * to be passed.
  */
  public SaamEvent(String eventSource,
    SaamTalker talker, int channel_ID){
    super(eventSource);
    this.talker=talker;
    this.channel_ID = channel_ID;
  }

  /**
   * Returns the ID of the channel associated with this event.
   * @return The ID of the channel associated with this event.
   */
   public int getChannel_ID(){
    return channel_ID;
   }

   /**
    * Sets the Channel ID for this event.
    * @param channel_ID The new Channel ID.
    */
   public void setChannel_ID(int channel_ID){
    this.channel_ID=channel_ID;
   }

   /**
    * Returns the SaamTalker associated with this event.
    * @return The SaamTalker associated with this event.
    */
   public SaamTalker getTalker(){
    return talker;
   }

   /**
```

```
  * Sets the SaamTalker for this event.
  * @param talker The new SaamTalker.
  */
 public void setTalker(SaamTalker talker){
  this.talker=talker;
 }

/**
 * Returns a <code>String</code> representation of this event.
 * @return The <code>String</code> representation of this event.
 */
 public String toString(){
   return "Channel_ID"+channel_ID+
     "\n  EventSource: "+getSource()+
     "\n  Talker: "+talker;

 }
}
```

```java
package saam.event;

import java.util.EventListener;

/**
 * A SaamListener merely listens for SaamEvents.
 */
public interface SaamListener extends EventListener{

   /**
    * This method is called by the Channel to pass event notifications
    * to the Objects that implement this interface.
    */
   public void receiveEvent(SaamEvent se);
}
```

```java
package saam.event;

import java.util.EventListener;

/**
 * A SaamListener merely listens for SaamEvents.
 */
public interface SaamListener extends EventListener{

   /**
    * This method is called by the Channel to pass event notifications
    * to the Objects that implement this interface.
    */
   public void receiveEvent(SaamEvent se);
}
```

```java
package saam.event;

/**
 * We have Listeners, why not Talkers?  Currently this interface is
merely
 * a designator for a category of Objects.  There are no methods
required
 * for its implementation.
 */
public interface SaamTalker{

}
```

```java
package saam.event;

import saam.net.SAAMPacket;
import saam.net.IPv6Address;
import saam.net.UDPHeader;

/**
 * A TransportInterfaceEvent is a simple SaamEvent that is intended to
 * be originated by the TransportInterface and destined for the
 * RoutingAlgorithm.
 */
public class TransportInterfaceEvent extends SaamEvent {

      /**
   * @serial
       */
   private int flowID;

  /**
   * @serial
   */
   private IPv6Address dest;

  /**
   * @serial
   */
   private UDPHeader udpHeader;

  /**
   * @serial
   */
   private SAAMPacket saamPacket;

   /**
    * Constructs a TransportInterfaceEvent with the attributes provided.
    * @param eventSource The location from which the event originated.
    * @param talker The Object that instantiated the event.
    * @param sourcePort The port from which the saamPacket originated.
    * @param flowID The flow on which this packet is to travel.
    * @param dest The IPv6Address of the destination for this
saamPacket.
    * @param udpHeader The UDPHeader associated with this packet.
    * @param saamPacket The SaamPacket that is to be sent.
    */
   public TransportInterfaceEvent(String eventSource,
     SaamTalker talker, int sourcePort, int flowID,
     IPv6Address dest, UDPHeader udpHeader,
     SAAMPacket saamPacket){
          super(eventSource,talker,sourcePort);
          this.flowID = flowID;
          this.dest = dest;
          this.udpHeader=udpHeader;
          this.saamPacket=saamPacket;
   }

  /**
   * Returns the flow ID associated with this event.
   * @return The flow ID associated with this event.
   */
   public int getFlowID(){
    return flowID;
```

```
    }

   /**
    * Returns the dest as an IPv6Address.
    * @return The dest as an IPv6Address.
    */
    public IPv6Address getDest(){
     return dest;
    }

   /**
    * Returns the UDPHeader associated with this event.
    * @return The UDPHeader associated with this event.
    */
    public UDPHeader getUDPHeader(){
     return udpHeader;
    }

   /**
    * Returns the SAAMPacket associated with this event.
    * @return The SAAMPacket associated with this event.
    */
    public SAAMPacket getSaamPacket(){
     return saamPacket;
    }

}
```

APPENDIX E. MESSAGE PACKAGE SOURCE CODE

```java
package saam.message;

import java.net.UnknownHostException;

import saam.net.IPv6Address;
import saam.util.Array;

/**
 * An ARPCacheEntry object contains an IPv6 address representing the
 * next hop and a byte representing the MAC address of the next hop.
 * The entry could be either an add or a remove depending on which
 * constructor is used.  An add is required to have both fields, whereas
 * a remove only requires the field that is used as the lookup key in
 * the associated table.
 * Entries can be retrieved as a byte array using the getBytes()
 * method.
 */
public class ARPCacheEntry extends Message{

    /**
     * The IPv6Address of the next hop for this Interface.
     */
    private IPv6Address nextHop;

    /**
     * The length of this entry varies depending on whether
     * the entry is intended to remove an entry from the associated
     * table or to add an entry into the associated table.  The default
     * length is that of an entry to be added to the table.
     */
    private short length = IPv6Address.length+1;

    /**
     * The MAC address of the NIC card on the next hop.
     */
    private byte nextMAC;

    /**
     * The ARPCacheEntry represented as a byte array.
     */
    private byte[] entry;

    /**
     * Creates a <em>remove</em> entry for the <em>ARP Cache</em> which
     * consists of an IPv6Address.
     * @param nextHop The IPv6Address of the next hop.
     */
    public ARPCacheEntry(IPv6Address nextHop){
      this.nextHop=nextHop;
      this.length=IPv6Address.length;
      entry = nextHop.getAddress();
    }

    /**
     * Creates an <em>add</em> entry for the <em>ARP Cache</em>.
     * @param nextHop The IPv6 address of the next hop.
     * @param nextMAC The MAC address of the next hop.
     * @param outboundMAC The MAC address of the outbound
     *                    interface.
     */
    public ARPCacheEntry(IPv6Address nextHop, byte nextMAC) {
```

```
      this.nextHop=nextHop;
      this.nextMAC = nextMAC;

      //now build the array for this entry
      entry = Array.concat(nextHop.getAddress(),nextMAC);
    }

    /**
     * Creates either an add or a remove entry for the <em>ARP
Cache</em>
     * depending on the length of the parameter.
     * @param entry The IPv6Address received from the subclass.
     */
    public ARPCacheEntry(byte[] entry){
      this.entry = entry;
      try{
        nextHop = new
IPv6Address(Array.getSubArray(entry,0,IPv6Address.length));
      }catch(UnknownHostException uhe){
        //do something with this
      }
      if(entry.length == IPv6Address.length){
            this.length = (short)entry.length;
      }else{
        nextMAC = entry[IPv6Address.length];
      }
    }
    /**
     * Returns the IPv6 address of the next hop associated with this
entry.
     * @return The IPv6 address of the next hop associated with this
entry.
     */
    public IPv6Address getNextHop(){
      return nextHop;
    }

    /**
     * Returns the MAC address of the NIC card on the next hop.
     * @return The MAC address of the NIC card on the next hop.
     */
    public byte getNextMAC(){
      return nextMAC;
    }

    /**
     * Returns The ARPCacheEntry as a byte array with
     *          the IPv6 address in the lowest order bytes.
     * @return The ARPCacheEntry as a byte array with
     *          the IPv6 address in the lowest order bytes.
     */
    public byte[] getBytes(){
      return entry;
    }

    /**
     * Returns the length of this entry.
     * @return The length of this entry.
     */
    public short length(){
```

```
      return length;
    }

    /**
     * Returns a <code>String</code> representation of this entry.
     * @return The <code>String</code> representation of this entry
     */
    public String toString() {
        return ("\nNext Hop: " + nextHop.toString() +
            (length==IPv6Address.length? "":"\nNext MAC:" + nextMAC));
    }
}
```

```
package saam.message;

import java.net.UnknownHostException;
import java.util.Vector;
import java.util.Enumeration;

import saam.net.*;
import saam.util.*;

/**
 * The DemoHello Message is intended to be sent from the DemoStation
 * and received by a ControlExecutive.  The purpose of this Message is
 * to transmit a set of InterfaceIDs to a router so the ControlExecutive
 * can stand up an Interface associated with each InterfaceID provided.
 */
public class DemoHello extends Message
{

  private Vector interfaceIDs;
  private byte[] bytes;

  /**
   * Constructs a DemoHello Message with the InterfaceIDs provided.
   * @param interfaceIDs The Vector of InterfaceIDs that will be
assigned to
   * the Interfaces on the destination router.
   */
  public DemoHello(Vector interfaceIDs){
    this.interfaceIDs = interfaceIDs;
    Enumeration e = interfaceIDs.elements();
    while(e.hasMoreElements()){
      InterfaceID myInterface = (InterfaceID)e.nextElement();

      //build the byte array
      bytes = Array.concat(bytes,myInterface.getBytes());
    }
  }

  /**
   * Constructs a DemoHello Message from a byte array that is assumed to
be
   * in the proper format for a DemoHello.  Presumably, this constructor
   * is called when the receiving PacketFactory gets the byte array that
   * represents this Message - a byte array that was presumably
generated
   * when the sender of this Message called the getBytes() method after
creating
   * this Message and before sending it.
   */
  public DemoHello(byte[] bytes){
    this.bytes = bytes;
    interfaceIDs = new Vector();
    int i = 0;
    while(i < bytes.length){
      try{
        interfaceIDs.addElement(new InterfaceID(
            Array.getSubArray(bytes, i, i+InterfaceID.length)));
        i+=InterfaceID.length;
      }catch(UnknownHostException uhe){}
    }
  }
```

```java
  /**
   * Returns the Vector of InterfaceIDs associated with this Message.
   * @return The Vector of InterfaceIDs associated with this Message.
   */
  public Vector getInterfaceIDs(){
    return interfaceIDs;
  }

  /**
   * Returns the length of this Message.
   * @return The length of this Message.
   */
  public short length(){
    try{
      return (short)bytes.length;
    }catch(NullPointerException npe){
      return 0;
    }
  }

  /**
   * Returns The byte array representation of this Message.
   * @return The byte array representation of this Message.
   */
  public byte[] getBytes(){
    return bytes;
  }

  /**
   * Returns a <code>String</code> representation of this Message.
   * @return The <code>String</code> representation of this Message
   */
  public String toString(){
    Enumeration e = interfaceIDs.elements();
    while(e.hasMoreElements()){
      e.nextElement().toString();
    }

    return("Hello:\n"+interfaceIDs.toString());
  }

}
```

```
package saam.message;

import java.net.UnknownHostException;
import java.net.InetAddress;
import saam.net.IPv6Address;
import saam.net.IPv4Address;
import saam.util.Array;

/**
 * An EmulationTableEntry object contains an IPv6 address representing
 * the next hop and an IPv4 address that maps to this IPv6 address.
 * The entry could be either an add or a remove depending on which
 * constructor is used.  An add is required to have both fields, whereas
 * a remove only requires the field that is used as the lookup key in
 * the associated table.
 * Entries can be retrieved as a byte array using the getBytes()
 * method.
 */
public class EmulationTableEntry extends Message{

      //default length
      private short length = IPv6Address.length + IPv4Address.length;

    /**
     * The IPv6 address associated with the next hop.
     */
    private IPv6Address nextHop;

    /**
     * The IPv4 address associated with the next hop.
     */
    private InetAddress v4;

    /**
     * The EmulationTableEntry represented as a byte array.
     */
    private byte[] entry;

    /**
     * Creates an entry for the <em>emulation table</em>.
     * @param nextHop The IPv6 address to be stored in the Emulation
Table.
     * @param v4 The IPv4 address to be stored in the Emulation Table.
     */
    public EmulationTableEntry(IPv6Address nextHop, InetAddress v4) {
      this.nextHop=nextHop;
      this.v4 = v4;

      //populate the entry array
      entry = Array.concat(nextHop.getAddress(),v4.getAddress());
    }

    /**
     * Creates an entry for the <em>emulation table</em> which
     * consists of an IPv6Address.
     * @param entry The IPv6Address received from the subclass.
     */
    public EmulationTableEntry(IPv6Address nextHop){
      this.nextHop=nextHop;
      length = IPv6Address.length;
      entry = nextHop.getAddress();
```

```
    }

    /**
     * Creates an entry for the <em>emulation table</em> which
     * consists of an IPv6Address.
     * @param entry The IPv6Address received from the subclass.
     */
    public EmulationTableEntry(byte[] entry){
      this.entry = entry;
      try{
        nextHop = new
IPv6Address(Array.getSubArray(entry,0,IPv6Address.length));
      }catch(UnknownHostException uhe){
        //do something with this
      }
      if(entry.length == IPv6Address.length){
            this.length = (short)entry.length;
      }else{
        byte[] v4Array = Array.getSubArray(
          entry,IPv6Address.length,entry.length);
        try{
          this.v4 =
            InetAddress.getByName(IPv4Address.toString(v4Array));
        }catch(UnknownHostException uhe){
          //do something with this
        }//try-catch
      }//if-else
    }//EmulationTableEntry

    /**
     * Returns the IPv6 address in this entry.
     * @return The IPv6Address used as a key in this entry.
     */
    public IPv6Address getIPv6Address(){
      return nextHop;
    }

    /**
     * Returns the IPv4 address in this entry.
     * @return The IPv4 address in this entry.
     */
    public InetAddress getIPv4Address(){
      return v4;
    }

    /**
     * Returns The EmulationTableEntry as a byte array with
     *          the IPv6 address in the lowest order bytes.
     * @return The EmulationTableEntry as a byte array with
     *          the IPv6 address in the lowest order bytes.
     */
    public byte[] getBytes(){
      return entry;
    }

    /**
     * Returns the length of this entry.
     * @return The length of this entry.
     */
    public short length(){
      return length;
```

```
    }

   /**
    * Returns a <code>String</code> representation of this Message.
    * @return The <code>String</code> representation of this Message
    */
    public String toString() {
        return ("Next Hop: " + nextHop.toString() +
             (length==IPv6Address.length? "":", " +
v4.getHostAddress()));
    }
}
```

```
package saam.message;

import java.net.UnknownHostException;
import saam.net.*;
import saam.util.*;


/**
 * An Object desiring to communicate within a SAAM network will call the
 * requestFlow method in the ControlExecutive.  The ControlExecutive
will
 * then construct a FlowRequest Message and send it to the server.
 */
public class FlowRequest extends Message{

  private IPv6Address source_interface = new IPv6Address();
  private IPv6Address destination_interface = new IPv6Address();
  private long time_stamp;
  private int requested_delay;
  private int requested_loss_rate;
  private int requested_throughput;

  private byte[] bytes;

  /**
   * No-Args constructor which constructs a FlowRequest using the
default
   * values for all fields. <p>
   * source_interface      = IPv6Address.DEFAULT_HOST;
   * destination_interface = IPv6Address.DEFAULT_HOST;
   * time_stamp            = 0;
   * requested_delay       = 0;
   * requested_loss_rate   = 0;
   * requested_throughput  = 0;
   */
  public FlowRequest(){
  }

  /**
   * Constructs a FlowRequest using the parameters supplied.
   * @param source_interface The IPv6Address of the source.
   * @param destination_interface The IPv6Address of the destination.
   * @param time_stamp The 8 byte time stamp.
   * @param requested_delay The maximum delay reuested.
   * @param requested_loss_rate The maximum loss rate requested.
   * @param requested_throughput The maximum throughput requested.
   */
  public FlowRequest(IPv6Address source_interface,
    IPv6Address destination_interface,
    long time_stamp,
    int requested_delay,
    int requested_loss_rate,
    int requested_throughput){

    //set all instance variables
    this.source_interface = source_interface;
    this.destination_interface = destination_interface;
    this.time_stamp = time_stamp;
    this.requested_delay = requested_delay;
    this.requested_loss_rate = requested_loss_rate;
    this.requested_throughput = requested_throughput;
```

```
    //build the byte array
    bytes = Array.concat(source_interface.getAddress(),
      destination_interface.getAddress());
    bytes = Array.concat(bytes,
      PrimitiveConversions.getBytes(time_stamp));
    bytes = Array.concat(bytes,
      PrimitiveConversions.getBytes(requested_delay));
    bytes = Array.concat(bytes,
      PrimitiveConversions.getBytes(requested_loss_rate));
    bytes = Array.concat(bytes,
      PrimitiveConversions.getBytes(requested_throughput));
  }

  /**
   * Construct this Message from a byte array that is presumed to
conform
   * to the proper format for this Message.  Presumably, this
constructor
   * is called when the receiving PacketFactory gets the byte array that
   * represents this Message - a byte array that was presumably
generated
   * when the sender of this Message called the getBytes() method after
creating
   * this Message and before sending it.
   */
  public FlowRequest(byte[] bytes)
    throws UnknownHostException{
    this.bytes = bytes;
    int pointer=0;
    try{
      source_interface = new IPv6Address(Array.
        getSubArray(bytes,pointer,IPv6Address.length));
      pointer += IPv6Address.length;
      destination_interface = new IPv6Address(Array.
        getSubArray(bytes,pointer,pointer+IPv6Address.length));
      pointer += IPv6Address.length;
      time_stamp = PrimitiveConversions.getLong(
        Array.getSubArray(bytes,pointer, pointer+8));
      pointer += 8;
      requested_delay = PrimitiveConversions.getInt(
        Array.getSubArray(bytes,pointer, pointer+4));
      pointer += 4;
      requested_loss_rate = PrimitiveConversions.getInt(
        Array.getSubArray(bytes,pointer, pointer+4));
      pointer += 4;
      requested_throughput = PrimitiveConversions.getInt(
        Array.getSubArray(bytes,pointer, pointer+4));
    }catch(UnknownHostException uhe){
      throw new UnknownHostException(uhe.toString());
    }
  }

  /**
   * Returns the IPv6Address of the source.
   * @return The IPv6Address of the source.
   */
  public IPv6Address getSourceInterface(){
    return source_interface;
  }
```

```
  /**
   * Returns the network address associated with the source IPv6Address
   * @return The network address associated with the source IPv6Address
   */
  public IPv6Address getSourceLink(){
    return source_interface.getNetworkAddress();
  }

  /**
   * Returns the IPv6Address of the destination.
   * @return The IPv6Address of the destination.
   */
  public IPv6Address getDestinationInterface(){
    return destination_interface;
  }

  /**
   * Returns the network address associated with the destination
IPv6Address
   * @return The network address associated with the destination
IPv6Address
   */
  public IPv6Address getDestinationLink(){
    return destination_interface.getNetworkAddress();
  }

  /**
   * Returns the 8 byte time stamp associated with this Message.
   * @return The 8 byte time stamp associated with this Message.
   */
  public long getTimeStamp(){
    return time_stamp;
  }

  /**
   * Returns the requested delay associated with this Message.
   * @return The requested delay associated with this Message.
   */
  public int getRequestedDelay(){
    return requested_delay;
  }

  /**
   * Returns the requested loss rate associated with this Message.
   * @return The requested loss rate associated with this Message.
   */
  public int getRequestedLossRate(){
    return requested_loss_rate;
  }

  /**
   * Returns the requested throughput associated with this Message.
   * @return The requested throughput associated with this Message.
   */
  public int getRequestedThroughput(){
    return requested_throughput;
  }

  /**
   * Returns The byte array representation of this Message.
   * @return The byte array representation of this Message.
```

```
   */
  public byte[] getBytes(){
    return bytes;
  }

  /**
   * Returns the length of this Message.
   * @return The length of this Message.
   */
  public short length(){
    try{
      return (short)bytes.length;
    }catch(NullPointerException npe){
      return 0;
    }
  }

  /**
   * Returns a <code>String</code> representation of this Message.
   * @return The <code>String</code> representation of this Message
   */
  public String toString(){
    return("Source: " + source_interface.toString() +
      ", Destination: " + destination_interface.toString() +
      ", TS: " + time_stamp + ", D: " +
      requested_delay + ", LR: " +
      requested_loss_rate + ", T: " +
      requested_throughput);
  }

}
```

```
package saam.message;

import java.net.UnknownHostException;
import saam.net.*;
import saam.util.*;


/**
 * A Response to a flow request simply contains the timestamp that was
 * sent with the corresponding FlowRequest, and the new flow id that has
 * been assigned to the Object requesting the flow.
 */
public class FlowResponse extends Message{

  private long time_stamp;
  private int flow_id;

  private byte[] bytes;

  /**
   * No-args constructor used by the server.
   */
  public FlowResponse(){}

  /**
   * Constructs a FlowResponse with the parameters supplied.
   * @param time_stamp The 8 byte time stamp associated with this
Message.
   * @param flow_ID The flow associated with this Message.
   */
  public FlowResponse(long time_stamp, int flow_id){

    this.time_stamp = time_stamp;
    this.flow_id = flow_id;

    bytes = Array.concat(
      PrimitiveConversions.getBytes(time_stamp),
      PrimitiveConversions.getBytes(flow_id));
  }

  /**
   * Construct this Message from a byte array that is presumed to
conform
   * to the proper format for this Message.  Presumably, this
constructor
   * is called when the receiving PacketFactory gets the byte array that
   * represents this Message - a byte array that was presumably
generated
   * when the sender of this Message called the getBytes() method after
creating
   * this Message and before sending it.
   */
  public FlowResponse(byte[] bytes)
    throws UnknownHostException{
    this.bytes = bytes;
    int pointer=0;
    time_stamp = PrimitiveConversions.getLong(
      Array.getSubArray(bytes,pointer, 8));
    pointer = pointer + 8;
    flow_id = PrimitiveConversions.getInt(
      Array.getSubArray(bytes,pointer, pointer+4));
```

```
  }

  /**
   * Returns the 8 byte time stamp associated with this Message.
   * @return The 8 byte time stamp associated with this Message.
   */
  public long getTimeStamp(){
    return time_stamp;
  }

  /**
   * Returns the flow ID associated with this event.
   * @return The flow ID associated with this event.
   */
  public int getFlowId(){
    return flow_id;
  }

  /**
   * Returns The byte array representation of this Message.
   * @return The byte array representation of this Message.
   */
  public byte[] getBytes(){
    return bytes;
  }

  /**
   * Returns the length of this Message.
   * @return The length of this Message.
   */
  public short length(){
    try{
      return (short)bytes.length;
    }catch(NullPointerException npe){
      return 0;
    }
  }

  /**
   * Returns a <code>String</code> representation of this Message.
   * @return The <code>String</code> representation of this Message
   */
  public String toString(){
    return("TimeStamp: " + time_stamp +
      " Flow id: " + flow_id);
  }
}
```

```java
package saam.message;

import saam.net.IPv6Address;
import saam.util.*;
import java.net.UnknownHostException;

/**
 * A FlowRoutingTableEntry object contains a 3 byte integer representing
 * the flow ID, a one byte service level, and an IPv6 address
representing
 * the next hop.<p>
 * The entry could be either an add or a remove depending on which
 * constructor is used.  An add is required to have all fields, whereas
 * a remove only requires the field that is used as the lookup key in
 * the associated table.
 * Entries can be retrieved as a byte array using the getBytes()
 * method.
 */
public class FlowRoutingTableEntry extends Message{


  /**
   * The number of bytes in a FlowRoutingTableEntry.
   */
    private short length = (short)(3+1+IPv6Address.length);

    /**
     * Only the three lowest-order bytes of this int value
     * are used.  Be sure to use values lower than
     * 2^25 (33,554,432) since only 24 bits are examined.
     */
    private int flowID = 0;
    private byte   sl;
    private IPv6Address nextHop;

    private byte[] entry = new byte[3];

    /**
     * Constructs an entry for the <em>Flow Routing Table</em>.
     * @param flowID The IPv4 address to be stored.
     */
    public FlowRoutingTableEntry(int flowID, byte sl,
                               IPv6Address nextHop) {
      this.flowID = flowID;
      this.sl = sl;
      this.nextHop = nextHop;

      //now build the array for this entry
      entry[2] = (byte)flowID;
      entry[1] = (byte)(flowID>>8);
      entry[0] = (byte)(flowID>>16);
      entry = Array.concat(entry,sl);
      entry = Array.concat(entry,
        Array.getSubArray(nextHop.getAddress(),0,16));
    }

    /**
     * Constructs an entry for the <em>Flow Routing Table</em>.
     * @param v6 The IPv6 address representing the lookup key
     * in the Emulation Table.
     */
```

```
public FlowRoutingTableEntry(int flowID) {
  this.flowID = flowID;
  entry = PrimitiveConversions.getBytes(flowID,3);
  this.length = 3;
}
/**
 * Constructs a FlowRoutingTableEntry from a byte array.
 * intended to be used by objects that receive routing
 * table updates over the network.
 * @param entry The byte array containing the properly
 *              ordered elements of a FlowRoutingTableEntry.
 *              i.e. flowID, SL, IPv6Address from lowest-
 *              ordered bits to highest in the entry array.
 */
public FlowRoutingTableEntry(byte[] entry) throws
  UnknownHostException{

    this.entry = entry;
    flowID = PrimitiveConversions.getInt(
      Array.getSubArray(entry,0,3));
    if(entry.length==3){
      this.length=3;
    }else{
      sl = entry[3];
      try{
        nextHop = new IPv6Address(
          Array.getSubArray(entry,4,entry.length));
      }catch(UnknownHostException uhe){
        throw new UnknownHostException(uhe.toString());
      }
    }
}

/**
 * Returns the Service Level as a byte.
 * @return The Service Level as a byte.
 */
public byte getSL(){
  return sl;
}

/**
 * Returns the next hop as an IPv6Address object.
 * @return The next hop as an IPv6Address object.
 */
public IPv6Address getNextHop(){
  return nextHop;
}

/**
 * Returns the flow ID associated with this Message
 * @return The flow ID associated with this Message
 */
public int getFlowID(){
  return flowID;
}

/**
 * Returns The FlowRoutingTableEntry as a byte array with
 *         the flow ID in the lowest order bytes.
 * @return The FlowRoutingTableEntry as a byte array with
```

```
 *          the flow ID in the lowest order bytes.
 */
public byte[] getBytes(){
  return entry;
}

/**
 * Returns a <code>String</code> representation of this entry
 * @return The <code>String</code> representation of this entry
 */
public String toString() {
    return ("FlowID: " + flowID +
      (length==3? ".":
      ", SL: "+(new Byte(sl)).toString()+
      ", Next Hop: " + nextHop.toString()));
}

/**
 * Returns the length of this entry.
 * @return The length of this entry.
 */
public short length(){
    return length;
}
}
```

```
package saam.message;

import java.net.UnknownHostException;
import java.util.Vector;
import java.util.Enumeration;

import saam.net.*;
import saam.util.*;

/**
 * The Hello Message is intended to be sent from the routers
 * and received by the server.  The purpose of this Message is
 * to transmit the set of InterfaceIDs associated with the interfaces
 * that have been instantiated on the sending router.
 */
public class Hello extends Message
{

  private Vector interfaceIDs;
  private byte[] bytes;

  /**
   * Constructs a Hello Message with the InterfaceIDs provided.
   * @param interfaceIDs The Vector of InterfaceIDs that will be
assigned to
   * the Interfaces on the destination router.
   */
  public Hello(Vector interfaceIDs){
    this.interfaceIDs = interfaceIDs;
    Enumeration e = interfaceIDs.elements();
    while(e.hasMoreElements()){
      InterfaceID myInterface = (InterfaceID)e.nextElement();
      bytes = Array.concat(bytes,myInterface.getBytes());
    }
  }

  /**
   * Construct this Message from a byte array that is presumed to
conform
   * to the proper format for this Message.  Presumably, this
constructor
   * is called when the receiving PacketFactory gets the byte array that
   * represents this Message - a byte array that was presumably
generated
   * when the sender of this Message called the getBytes() method after
creating
   * this Message and before sending it.
   */
  public Hello(byte[] bytes){
    this.bytes = bytes;
    interfaceIDs = new Vector();
    IPv6Address myInterfaceAddress;
    int i = 0;
    int bandwidth = 0;
    byte mac = 0;
    while(i < bytes.length){
      try{
        myInterfaceAddress = new IPv6Address(
          Array.getSubArray(bytes, i, i+IPv6Address.length));
        i = i + IPv6Address.length;
        bandwidth = PrimitiveConversions.getInt(
```

```
            Array.getSubArray(bytes, i, i+4));
        i += 4;
        mac = (byte)PrimitiveConversions.getInt(
            Array.getSubArray(bytes, i, i+1));
        i += 1;
        interfaceIDs.addElement(new InterfaceID(
            myInterfaceAddress,bandwidth,mac));
      }catch(UnknownHostException uhe){}
    }
  }

  /**
   * Returns the Vector of InterfaceIDs associated with this Message.
   * @return The Vector of InterfaceIDs associated with this Message.
   */
  public Vector getInterfaceIDs(){
    return interfaceIDs;
  }

  /**
   * Returns The byte array representation of this Message.
   * @return The byte array representation of this Message.
   */
  public byte[] getBytes(){
    return bytes;
  }

  /**
   * Returns the length of this Message.
   * @return The length of this Message.
   */
  public short length(){
    try{
      return (short)bytes.length;
    }catch(NullPointerException npe){
      return 0;
    }
  }

  /**
   * Returns a <code>String</code> representation of this Message.
   * @return The <code>String</code> representation of this Message
   */
  public String toString(){
    Enumeration e = interfaceIDs.elements();
    while(e.hasMoreElements()){
      e.nextElement().toString();
    }

    return("Hello:\n"+interfaceIDs.toString());
  }
}
```

```
package saam.message;

import java.net.UnknownHostException;
import java.util.*;
import saam.net.*;
import saam.util.*;

/**
 * An InterfaceID provides all necessary information to identify an
Interface.
 * Every Interface has an IPv6Address, a MAC address, and an associated
 * bandwidth.
 */
public class InterfaceID extends Message
{

  public static final short length = IPv6Address.length + 1 + 4;

  private IPv6Address v6;
  private int bandwidth = 10000; //10 Mbps ethernet
  private byte mac = 0;

  private byte[] id;

  /**
   * Constructs an InterfaceID with the IPv6Address and MAC provided and
   * the default value for the bandwidth.
   * @param v6 The IPv6Address of the interface.
   * @param mac The MAC address of the interface.
   */
  public InterfaceID(IPv6Address v6, byte mac){
    this.v6 = v6;
    this.mac = mac;

    id =
Array.concat(v6.getAddress(),PrimitiveConversions.getBytes(bandwidth));
    id = Array.concat(id,mac);
  }

  /**
   * Constructs an InterfaceID with the IPv6Address and bandwidth
provided and
   * the default value for the MAC.
   * @param v6 The IPv6Address of the interface.
   * @param bandwidth The bandwidth of the link the interface is
connected to.
   */
  public InterfaceID(IPv6Address v6, int bandwidth){
    this.v6 = v6;
    this.bandwidth = bandwidth;

    id =
Array.concat(v6.getAddress(),PrimitiveConversions.getBytes(bandwidth));
    id = Array.concat(id,mac);
  }

  /**
   * Constructs an InterfaceID with the values provided.
   * @param v6 The IPv6Address of the interface.
   * @param bandwidth The bandwidth of the link the interface is
connected to.
```

```
   * @param mac The MAC address of the interface.
   */
  public InterfaceID(IPv6Address v6, int bandwidth, byte mac){
    this.v6 = v6;
    this.bandwidth = bandwidth;
    this.mac = mac;

    id =
Array.concat(v6.getAddress(),PrimitiveConversions.getBytes(bandwidth));
    id = Array.concat(id,mac);
  }

  /**
   * Construct this Message from a byte array that is presumed to
conform
   * to the proper format for this Message.  Presumably, this
constructor
   * is called when the receiving PacketFactory gets the byte array that
   * represents this Message - a byte array that was presumably
generated
   * when the sender of this Message called the getBytes() method after
creating
   * this Message and before sending it.
   */
  public InterfaceID(byte[] id) throws UnknownHostException{
    this.id=id;
    try{
      v6 = new IPv6Address(
        Array.getSubArray(id,0,IPv6Address.length));
    }catch(UnknownHostException uhe){
      throw new UnknownHostException(uhe.toString());
    }
    bandwidth = PrimitiveConversions.getInt(Array.getSubArray(id,

IPv6Address.length,IPv6Address.length+4));

    mac = id[IPv6Address.length+4];
  }

  /**
   * Returns the IPv6Address associated with the Interface.
   * @return The IPv6Address associated with the Interface.
   */
  public IPv6Address getIPv6(){
    return v6;
  }

  /**
   * Returns the bandwidth associated with the Interface.
   * @return The bandwidth associated with the Interface.
   */
  public int getBandwidth(){
    return bandwidth;
  }

  /**
   * Returns the MAC address associated with the Interface.
   * @return The MAC address associated with the Interface.
   */
  public byte getMAC(){
    return mac;
```

```
  }

  /**
   * Returns The byte array representation of this Message.
   * @return The byte array representation of this Message.
   */
  public byte[] getBytes(){
    return id;
  }

  /**
   * Compares interfaceIDs by comparing the IPv6Addresses and the
   * MAC addresses.
   */
  public boolean equals(InterfaceID id){
    return v6.equals(id.getIPv6())&&
      mac==id.getMAC();
  }

  /**
   * Returns the length of this Message.
   * @return The length of this Message.
   */
  public short length(){
    return length;
  }

  /**
   * Returns a <code>String</code> representation of this Message.
   * @return The <code>String</code> representation of this Message
   */
  public String toString(){
    return("IP: "+v6.toString()+", bandwidth: "+bandwidth+", MAC:
"+mac+"\n");
  }
}
```

```
package saam.message;

import java.net.UnknownHostException;
import saam.net.*;
import saam.util.*;

/**
 * LinkStateAdvertisement are sent by the routers in a SAAM network to
 * keep the server abreast of their state.  The server makes routing
decisions
 * and adjustments to the network based on the LinkStateAdvertisements
it
 * receives.
 */
public class LinkStateAdvertisement extends Message{

  public static final int length = 29;

  private IPv6Address myIPv6 = new IPv6Address();
  private byte ServiceLevel;
  private int Delay = -10; // milliseconds
  private int LossRate = -100; // %
  private int Utilization = 110; // %*10 (throughput=utilization*(BW
allocated for SLP))

  private byte[] payload;

  /**
   * Used by the Server.
   */
  public LinkStateAdvertisement(){}

  /**
   * Used by the LsaGenerator.
   */
  public LinkStateAdvertisement(byte serviceLevel){
    ServiceLevel = serviceLevel;
  }

  /**
   * Constructs a LinkStateAdvertisement with the values provided.
   * @param myIPv6 The IPv6Address of the sender.
   * @param ServiceLevel The service level being reported on.
   * @param Delay The delay associated with the service level.
   * @param LossRate The loss rate associated with the service level.
   * @param Utilization The bandwidth utilization associated with the
service
   *        level.
   */
  public LinkStateAdvertisement(IPv6Address myIPv6, byte ServiceLevel,
                                 int Delay, int LossRate, int
Utilization){

    this.myIPv6 = myIPv6;
    this.ServiceLevel = ServiceLevel;
    this.Delay = Delay;
    this.LossRate = LossRate;
    this.Utilization = Utilization;

    payload = Array.concat(payload,myIPv6.getAddress());
    payload = Array.concat(payload,ServiceLevel);
```

```
    payload = Array.concat(payload,(new Integer(Delay)).byteValue());
    payload = Array.concat(payload,(new Integer(LossRate)).byteValue());
    payload = Array.concat(payload,(new
Integer(Utilization)).byteValue());
  }

  /**
   * Construct this Message from a byte array that is presumed to
conform
   * to the proper format for this Message.  Presumably, this
constructor
   * is called when the receiving PacketFactory gets the byte array that
   * represents this Message - a byte array that was presumably
generated
   * when the sender of this Message called the getBytes() method after
creating
   * this Message and before sending it.
   */
  public LinkStateAdvertisement(byte[] payload) throws
UnknownHostException{
    this.payload = payload;
    int pointer=0;
    try{
      myIPv6 = new IPv6Address(Array.

getSubArray(payload,pointer,IPv6Address.length));
      pointer = IPv6Address.length;
      ServiceLevel = payload[pointer];
      pointer = pointer + 1;
      Delay = PrimitiveConversions.getInt(
        Array.getSubArray(payload,pointer, 4));
      pointer = pointer + 4;
      LossRate = PrimitiveConversions.getInt(
        Array.getSubArray(payload,pointer, 4));
      pointer = pointer + 4;
      Utilization = PrimitiveConversions.getInt(
        Array.getSubArray(payload,pointer, 4));
    }catch(UnknownHostException uhe){
      throw new UnknownHostException(uhe.toString());
    }
  }

  /**
   * Returns the IPv6Address of the sender.
   * @return The IPv6Address of the sender.
   */
  public IPv6Address getMyIPv6(){
    return myIPv6;
  }

  /**
   * Returns the service level being reported on.
   * @return The service level being reported on.
   */
  public byte getServiceLevel(){
    return ServiceLevel;
  }

  /**
   * Returns the delay associated with the service level.
   * @return The delay associated with the service level.
```

```
   */
  public int getDelay(){
    return Delay;
  }

  /**
   * Returns the packet loss rate associated with the service level.
   * @return The packet loss rate associated with the service level.
   */
  public int getLossRate(){
    return LossRate;
  }

  /**
   * Returns the bandwidth utilization associated with the service
level.
   * @return The bandwidth utilization associated with the service
level.
   */
  public int getUtilization(){
    return Utilization;
  }

  /**
   * Returns The byte array representation of this Message.
   * @return The byte array representation of this Message.
   */
  public byte[] getBytes(){
    return payload;
  }

  /**
   * Returns the length of this Message.
   * @return The length of this Message.
   */
  public short length(){
    return length;
  }

  /**
   * Returns a <code>String</code> representation of this Message.
   * @return The <code>String</code> representation of this Message
   */
  public String toString(){
    return("Source: " + myIPv6.toString() + ", SL: " + ServiceLevel
      + ", Delay: " + Delay + ", LossRate: " + LossRate
      + ", Utilization: " + Utilization);
  }

}
```

```java
package saam.message;

/**
 * The Message class provides a convenient way for Objects to
communicate
 * with one another over a SAAM network.  The standard JDK does not
currently
 * provide a means to serialize objects over UDP.  This class does just
that.
 * Subclasses need to be written as follows to enable this
functionality:<p>
 * 1. Provide a constructor that accepts a byte array as its only
parameter.
 * 2. Override the getBytes method in such a way that it returns a byte
array
 *    that contains the values of the variables to be transferred.
 * 3. Ensure that the constructor mentioned above is set up to properly
 *    parse the byte array and rebuild the variables as they were
originally.
 * 4. Ensure that the length method returns the actual length of the
byte array.
 */
public abstract class Message{

  private static byte type = 1;

  /**
   * Returns the type value.
   * @return The type value.
   */
  public byte getType(){
    return type;
  }

  /**
   * Returns the length of this Message.
   * @return The length of this Message.
   */
  public abstract short length();

  /**
   * Returns The byte array representation of this Message.
   * @return The byte array representation of this Message.
   */
  public abstract byte[] getBytes();

  /**
   * Returns a <code>String</code> representation of this Message.
   * @return The <code>String</code> representation of this Message
   */
  public String toString(){
    return "Message";
  }
}
```

```java
package saam.message;

/**
 * In order for a Message to be delivered on a SAAM network, there must
 * be a MessageProcessor on the router that is registered with the
 * ControlExecutive to process Messages of that type.  To become a
 * MessageProcessor that can actually process messages, it is not enough
 * to simply implement this interface.  Objects implementing this
interface
 * must also provide a call to the ControlExecutive's
registerMessageProcessor
 * method, passing itself as a parameter.  The ControlExecutive will
then
 * make a call back to the registrant to retrieve the array of Messages
this
 * MessageProcessor would like to process.  Therefor, an additional
requirement
 * of implementors of this interface is that they must supply a String
array
 * that contains the String names (fully qualified classnames) of the
Messages
 * it would like to process.  The ControlExecutive will then know that
when a
 * Message of that type arrives, it should be sent to this processor.
 */
public interface MessageProcessor{

  /*
    An implementation of this interface would also include an array of
    Strings that describe the Messages that implementation desires to
    process as follows:
      String[] messagesIProcess =
        {"saam.message.SomeMessage","saam.message.AnotherMessage"};

    The implementation would also include a call to the
ControlExecutive's
    registerMessageProcessor method as follows:
      controlExec.RegisterMessageProcessor(this);
  */

  /**
   * This method contains the logic needed by the implementor of this
   * interface to process the Messages it is registered to process.
   * @param message The subclass of saam.message.Message to be
processed.
   */
      public void processMessage(Message message);

  /**
   * Returns the array of Strings representing the class
   * names of the messages this Object will register to process.
   * @return The array of Strings representing the class names
   * of the messages this Object will register to process.
   */
  public String[] getMessageTypes();
}
```

```java
package saam.message;

import java.net.UnknownHostException;
import java.net.InetAddress;
import saam.net.IPv4Address;
import saam.net.IPv6Address;
import saam.util.Array;

/**
 * A ServerID provides all necessary information to identify a Server.
 * Every Server has an IPv6Address and an associated InetAddress
 * (IPv4 address).
 */
public class ServerID extends Message{

  /**
   * The length of the byte array that contains the state of this
Message.
   */
  public static final short length =
    IPv6Address.length + IPv4Address.length;

  private IPv6Address v6;
  private InetAddress v4;

  private byte[] id;

  /**
   * Constructs an ServerID with the values provided.
   * @param v6 The IPv6Address of the interface.
   * @param v4 The IPv4 address of the interface.
   */
  public ServerID(IPv6Address v6, InetAddress v4){
    this.v6 = v6;
    this.v4 = v4;

    id = Array.concat(id,v6.getAddress());
    id = Array.concat(id,v4.getAddress());
  }

  /**
   * Construct this Message from a byte array that is presumed to
conform
   * to the proper format for this Message.  Presumably, this
constructor
   * is called when the receiving PacketFactory gets the byte array that
   * represents this Message - a byte array that was presumably
generated
   * when the sender of this Message called the getBytes() method after
creating
   * this Message and before sending it.
   */
  public ServerID(byte[] id) throws UnknownHostException{
    this.id=id;
    try{
      v6 = new IPv6Address(
        Array.getSubArray(id,0,IPv6Address.length));
      v4 = InetAddress.getByName(IPv4Address.toString(
        Array.getSubArray(id,IPv6Address.length,id.length)));
    }catch(UnknownHostException uhe){
      throw new UnknownHostException(uhe.toString());
```

```
    }
  }

  /**
   * Returns the IPv6Address associated with the Server.
   * @return The IPv6Address associated with the Server.
   */
  public IPv6Address getIPv6(){
    return v6;
  }

  /**
   * Returns the IPv4 address associated with the Server.
   * @return The IPv4 address associated with the Server.
   */
  public InetAddress getIPv4(){
    return v4;
  }

  /**
   * Returns The byte array representation of this Message.
   * @return The byte array representation of this Message.
   */
  public byte[] getBytes(){
    return id;
  }

  /**
   * Returns the length of this Message.
   * @return The length of this Message.
   */
  public short length(){
    return length;
  }

  /**
   * Returns a <code>String</code> representation of this Message.
   * @return The <code>String</code> representation of this Message
   */
  public String toString(){
    return("IPv6: "+v6.toString()+", IPv4: "+v4.getHostAddress());
  }
}
```

APPENDIX F. NET PACKAGE SOURCE CODE

```
package saam.net;

import java.util.StringTokenizer;
import java.net.UnknownHostException;

/**
 * The java.net.InetAddress class does not allow access to its
constructors,
 * nor does it allow subclassing.  So, if we have byte array that we
wish to
 * use to construct an InetAddress, we must come up with an alternate
method.
 * Since InetAddress.getByName() returns an InetAddress if provided with
a
 * String representation of the address, we need a static method that
converts
 * a byte array into a String representation of an InetAddress.  This
class
 * provides that method.
 */

public class IPv4Address {

   /**
    * The number of bytes in an IPv4Address.
    */
   public static final int length = 4;

        /**
         * The String representation of the IPv4Address.
         * @param address The IPv4Address represented as a byte array.
         * @return the String representation of the IPv4Address.
    */
        public static String toString(byte[] address) throws
UnknownHostException{
                StringBuffer buf = new StringBuffer();
                for(int i=0;i<address.length-1;i++){
                        //This is where the promotion occurs.  address
contains bytes,
                        //so we promote them to ints using the bitwise-and
operator.
                        //This enables us to display the value as an unsigned
byte.
                        buf.append((address[i]&0xFF)+".");
                }
                buf.append(address[address.length-1]&0xFF);
                return new String(buf);
        }
}
```

```
package saam.net;

import java.util.StringTokenizer;
import java.net.UnknownHostException;

/**
 * This class is loosely based on the java.net.InetAddress class.
 * Methods are designed to recieve parameters of the same types and
 * to return the same data types as InetAddress.
 */

public class IPv6Address {

  public static final String DEFAULT_HOST =
      "0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0";
  /**
   * The number of bytes in an IPv6Address.
   */
  public static final int length = 16;

  private static final int subnetMask = 4;

      /**
       * The 16 byte IPv6 address
       */
      private byte[] address = new byte[length];

      /**
       * The no-args constructor sets the address to local host
       * This may not be the correct IPv6 localhost address, but
       * it looks close.
   */
      public IPv6Address() {

            //set default as localhost
            String host=DEFAULT_HOST;

            StringTokenizer parser = new StringTokenizer(host,".");
            for(int i=0;parser.hasMoreTokens();i++){
                  short shortVal = (new Short(
                              parser.nextToken())).shortValue();
                  //casting a short to a byte enables you to store an
                  //unsigned byte into a Java byte.  The trick in
displaying
                  //this value is promoting it first.  See toString()
below.
                  address[i]= (byte)shortVal;
            }
      }

      /**
       * Constructs an IPv6 address with address = addr
       * @param addr The 16 byte IPv6 address
       */
      public IPv6Address(byte[] addr) throws UnknownHostException{
    if(addr.length!=length) {
      throw new UnknownHostException("Wrong number of bytes");
    }else{
            address = addr;
    }
      }
```

```
	/**
	 * Retrieves the IPv6 address as a 16 byte array.
	 * @return The IPv6 address
	 */
	public byte[] getAddress(){
		return address;
	}

	/**
	 * This static method allows the creation of an IPv6Address via
	 * the String representation of the address.  It is currently not
	 * capable of receiving a String representation of the host name.
	 * @param host A String representation of the IPv6 address.
	 * @return An IPv6Address with address = host.
	 */
	public static IPv6Address getByName(String host) throws
UnknownHostException{

		byte[] ipv6 = new byte[length];

		StringTokenizer parser = new StringTokenizer(host,".");
		if (parser.countTokens()!=length){
			throw new UnknownHostException("wrong number of octets
in host");
		}
		for(int i=0;parser.hasMoreTokens();i++){
			try{
			short shortVal = (new Short(
					parser.nextToken())).shortValue();
				if (shortVal>255){
				throw new UnknownHostException("value in host
too large: "+shortVal);
				}else if (shortVal<0){
				throw new UnknownHostException("value in host
too small: "+shortVal);
			}
			ipv6[i]= (byte)shortVal;
			}catch(NumberFormatException e){
			throw new UnknownHostException("invalid value in
host");
			}
		}
		IPv6Address addr = new IPv6Address();
		addr.address = ipv6;
		return addr;
	}

	/**
	 * Determines whether the two IPv6Address objects are equal
	 * criteria: If the byte arrays representing the objects are the
same,
	 * the objects are equal.
	 */
	public boolean equals(IPv6Address v6){
    byte[] in = v6.getAddress();
    try{
      for(int i=0; i<address.length; i++){
        if(address[i]!=in[i]) return false;
      }
      return true;
```

```java
    }catch(Exception e){
      return false;
    }
      }

  /**
   * Returns the network IPv6Address associated with this IPv6Address.
   * @return The network IPv6Address associated with this IPv6Address.
   */
  public IPv6Address getNetworkAddress(){
    byte[] networkAddress = new byte[length];
    IPv6Address netAddress = null;
    for (int index = 0;index < subnetMask; index++){
      networkAddress[index] = address[index];
    }
    for (int index = subnetMask;index < length; index++){
      networkAddress[index] = 0;
    }
    try {
      netAddress = new IPv6Address(networkAddress);
    } catch (UnknownHostException uhe) {
      System.err.println("IPv6Address: getNetworkAddress: " + uhe);
    }
    return netAddress;
  }

  /**
   * The String representation of the IPv6Address.
   * @return the String representation of the IPv6Address.
   */
  public String toString(){
    StringBuffer buf = new StringBuffer();
    for(int i=0;i<address.length-1;i++){
      //This is where the promotion occurs.  address contains bytes,
      //so we promote them to ints using the bitwise and operator.
      //This enables us to display the value as an unsigned byte.
      buf.append((address[i]&0xFF)+".");
    }
    buf.append(address[address.length-1]&0xFF);
    return new String(buf);
  }
}
```

```java
package saam.net;

import java.net.UnknownHostException;
import saam.util.Array;
import saam.util.PrimitiveConversions;

/**
 * This class constructs a standard IPv6 header.
 * An IPv6 Header is an element within every IPv6 packet that flows
 * through the SAAM architecture.  Instantiation of this object
 * constructs a 40-byte array that can be used for transport.
 * The class contains several methods for accessing each element of
 * the IPv6 header.
 */
public class IPv6Header {

  /**
   * The number of bytes in an IPv6Header.
   */
  public static final byte length = 40;

  /**
   * @serial
   */
     private byte      versionAndPriority;

  /**
   * @serial
   */
     private int               flowLabel;

  /**
   * @serial
   */
     private short           payloadLength;

  /**
   * @serial
   */
     private byte            nextHeader;

  /**
   * @serial
   */
     private byte            hopLimit = 3;

  /**
   * @serial
   */
     private IPv6Address     source, dest;

     private byte[]          v6Header = new byte[length];

     /**
      * Constructs an IPv6Header with values set according
      * to the params.
  * @param versionAndPriority The version and priority field.
      * @param flowLabel the 24-bit flow id (be sure this value does
      * not exceed the maximum value that 24 bits can contain.
  * @param payloadLength The length of the payload associated with this
  * header.
```

```
 * @param nextHeader The next header.
 * @param hopLimit The maximum number of hops this packet is allowed.
     * @param source The IPv6Address of the source.
     * @param dest The IPv6Address of the destination.
     */
    public IPv6Header(byte versionAndPriority, int flowLabel,
  short payloadLength, byte nextHeader, byte hopLimit,
  IPv6Address source, IPv6Address dest){

  this.versionAndPriority=versionAndPriority;
          //since the flowLabel is only 24 bits, we convert the
          //32-bit int to a 3-byte array
          this.flowLabel = flowLabel;
  this.payloadLength=payloadLength;
  this.nextHeader=nextHeader;
  this.hopLimit=hopLimit;
          this.source=source;
          this.dest=dest;
          setArray();
    }

/**
 * Constructs an IPv6Header with version=0, priority=0,
 * nextHeader=0, hopLimit=3, and the other values set according
 * to the params.  This constructor simply speeds the creation
 * process by using default values for fields that may not be
 * of concern to the user.
 * @param flowLabel the 24-bit flow id (be sure this value does
 * not exceed the maximum value that 24 bits can contain.
 * @param source The IPv6Address of the source.
 * @param dest The IPv6Address of the destination.
 */
public IPv6Header(int flowLabel, IPv6Address dest){

  //since the flowLabel is only 24 bits, we convert the
  //32-bit int to a 3-byte array
  this.flowLabel = flowLabel;
  this.source=new IPv6Address();
  this.dest=dest;
  setArray();
}

/**
 * A private method used internally to assign the values to the byte
array
 * that represents this header.
 */
    private void setArray(){
  v6Header = Array.concat(versionAndPriority,
    PrimitiveConversions.getBytes(flowLabel,3));
  v6Header = Array.concat(
    v6Header,PrimitiveConversions.getBytes(payloadLength));
  v6Header = Array.concat(v6Header,nextHeader);
  v6Header = Array.concat(v6Header,hopLimit);
  v6Header = Array.concat(v6Header,source.getAddress());
  v6Header = Array.concat(v6Header,dest.getAddress());
    }

/**
 * Sets the source field in this IPv6Header.
 * @param The new source address.
```

```
 */
    public void setSource(IPv6Address source){
          this.source=source;
          setArray();
    }

    /**
     * This constructor is used to construct an IPv6Header given
     * a 40-byte array containing properly ordered fields of a
     * standard IPv6 header.
     */
    public IPv6Header(byte[] data) throws UnknownHostException{
  v6Header=data;
          versionAndPriority = data[0];
  flowLabel = PrimitiveConversions.getInt(
    Array.getSubArray(data,1,4));
  payloadLength = PrimitiveConversions.getShort(
    Array.getSubArray(data,4,6));
          nextHeader    = data[6];
          hopLimit      = data[7];

          try{
                source = new IPv6Address(Array.getSubArray(
      data,8,8+IPv6Address.length));
                dest = new IPv6Address(Array.getSubArray(
      data,8+IPv6Address.length,8+(2*IPv6Address.length)));
          }catch(UnknownHostException uhe){
                throw new UnknownHostException(uhe.toString());
          }
    }

    /**
     * Dynamically sets the payload length.  Intended to be
     * invoked when a packet destined for the SAAM architecture
     * is constructed.
     * @param length The length of the IPv6 packet's payload.
     */
    public void setPayloadLength(short length){
  payloadLength = length;
    }

    /**
     * Returns the 40-byte IPv6Header.
     * @return The IPv6Header as a byte array of 40 bytes.
     */
    public byte[] getHeader(){
          return v6Header;
    }

/**
 * Returns the flow label of this IPv6Address as an int.
 * @return The flow label of this IPv6Address as an int.
 */
public int getFlowLabel(){
  return flowLabel;
}

    /**
     * Returns the payload length as a short.
     * @return The payload length as a short.
     */
```

```
    public short getPayloadLength(){
   return payloadLength;
    }

    /**
     * Returns the source as an IPv6Address.
     * @return The source as an IPv6Address.
     */
    public IPv6Address getSource(){
          return source;
    }

    /**
     * Returns the dest as an IPv6Address.
     * @return The dest as an IPv6Address.
     */
    public IPv6Address getDest(){
          return dest;
    }

  /**
   * Returns a <code>String</code> representation of this object
   * @return The <code>String</code> representation of this object
   */
    public String toString(){
          return
                "\nVer/Pri: "+versionAndPriority+
                "\nFlow:    "+flowLabel+
                "\nPyld ln:"+payloadLength+
                "\nNextHdr: "+nextHeader+
                "\nHopLim:  "+hopLimit+
                "\nSource:  "+source+
                "\nDest:    "+dest;
    }
}
```

```
package saam.net;

import java.net.UnknownHostException;
import saam.util.Array;

/**
 * An IPv6Packet contains an IPv6Header and a payload and methods to
access
 * both.
 */
public class IPv6Packet {

      private IPv6Header      v6header;
      private byte[] payload;
  private UDPHeader udpHeader;

      private byte[]          packet;

  /**
   * Constructs an IPv6Packet with the fields supplied.
   * @param v6header The properly formatted IPv6Header.
   * @param udpHeader The UDPHeader associated with this IPv6Packet.
   * @param payload A byte array containing the information the user
   *        wishes to transmit accross the network.
   */
  public IPv6Packet(IPv6Header v6header, UDPHeader udpHeader,
       byte[] payload){
    this.v6header=v6header;
    this.udpHeader=udpHeader;
    this.payload = payload;
    packet = Array.concat(v6header.getHeader(),udpHeader.getHeader());
    packet = Array.concat(packet,payload);
  }

      /**
       * This constructor is used to construct a packet given
       * a properly formatted byte array.
       */
      public IPv6Packet(byte[] packet) throws UnknownHostException{

    this.packet=packet;
            byte[] headerArray =
      Array.getSubArray(packet,0,IPv6Header.length);
            try{
                  v6header = new IPv6Header(headerArray);
            }catch(UnknownHostException uhe){
                  throw new UnknownHostException(uhe.toString());
            }
    udpHeader = new UDPHeader(Array.getSubArray(
      packet,
      IPv6Header.length,
      IPv6Header.length+UDPHeader.length));
    payload =
Array.getSubArray(packet,IPv6Header.length+UDPHeader.length,
      packet.length);
  }

  /**
   * Returns this IPv6Packet as a byte array.
   * @return This IPv6Packet as a byte array.
   */
```

```
  public byte[] getBytes(){
    return packet;
  }

  /**
   * Returns the v6header of this IPv6Packet.
   * @return The v6header of this IPv6Packet.
   */
  public IPv6Header getHeader(){
    return v6header;
  }

  /**
   * Sets the v6header of this IPv6Packet.
   * @param The new v6header for this IPv6Packet.
   */
  public void setHeader(IPv6Header v6header){
    this.v6header=v6header;
    packet = Array.concat(v6header.getHeader(),udpHeader.getHeader());
    packet = Array.concat(packet,payload);
  }

  /**
   * Returns the UDPHeader of this IPv6Packet.
   * @return The UDPHeader of this IPv6Packet.
   */
  public UDPHeader getUDPHeader(){
    return udpHeader;
  }

  /**
   * Returns the payload of this IPv6Packet.
   * @return The payload of this IPv6Packet.
   */
  public byte[] getPayload(){
    return payload;
  }

  /**
   * Returns a <code>String</code> representation of this object
   * @return The <code>String</code> representation of this object
   */
  public String toString(){
    return v6header.toString()+"\n"+udpHeader.toString();
  }
}
```

```
package saam.net;

import java.net.UnknownHostException;
import saam.util.Array;
import saam.util.PrimitiveConversions;

/**
 * A SAAMHeader contains an eight byte time stamp and a one byte field
 * containing the number of updates that are in the payload portion of
 * the SAAMPacket to which this header belongs.
 */
public class SAAMHeader {

  /**
   * This variable defines the number of bytes in the
   * time stamp field
   */
  public static final int timeStampLength = 8;

  /**
   * This variable defines the number of bytes in the
   * numberOfUpdates field
   */
  public static final int numberOfUpdatesLength = 1;

  /**
   * The length of this header in bytes.
   */
  public static final int length = numberOfUpdatesLength +
timeStampLength;

  /**
   * Represents the time the SAAM packet leaves the source.
   * This value should be set when the packet is sent.
   */
  private byte[]    timeStamp;

  /**
   * The numberOfUpdatesLength of SAAM Packet this is.
   */
      private byte      numberOfUpdates;

  /**
   * The byte-array representation of this header
   */
  private byte[]    header = new byte[length];


  /**
   * Constructs a SAAM header given the values that will
   * be contained within the header.
   * @param type Which type of SAAM packet this is.  Currently,
   *             this is contained in a single byte.
   * @param timeStamp The time associated with this packet.
   */
  public SAAMHeader(long timeStamp, byte numberOfUpdates){

    this.timeStamp = PrimitiveConversions.getBytes(timeStamp);
    this.numberOfUpdates = numberOfUpdates;

    header = Array.concat(this.timeStamp,numberOfUpdates);
```

```java
    }

    /**
     * Constructs a SAAMHeader from a byte array containing
     * the properly formatted SAAMHeader fields.
     * @param header A byte array containing the properly
     *        formatted SAAMHeader fields.
     */
    public SAAMHeader(byte[] header) throws UnknownHostException{
      this.header=header;
      timeStamp = Array.getSubArray(header,0,timeStampLength);
      numberOfUpdates = header[timeStampLength];
    }

    /**
     * Returns this SAAMHeader as a byte array.
     * @return This SAAMHeader as a byte array.
     */
    public byte[] getHeader(){
      return header;
    }

    /**
     * Returns the time stamp.
     * @return The time stamp.
     */
    public long getTimeStamp(){
      return PrimitiveConversions.getLong(timeStamp);
    }

    /**
     * Returns the numberOfUpdates in this SAAM packet.
     * @return The numberOfUpdates in this SAAM packet.
     */
    public byte getNumberOfUpdates(){
      return numberOfUpdates;
    }

    /**
     * Returns the String representation of this SAAMHeader.
     * @return The String representation of this SAAMHeader.
     */
    public String toString(){
      return "TimeStamp: "+getTimeStamp()+
        "\nNumber of Updates: "+numberOfUpdates;
    }//toString()
}
```

```java
package saam.net;

import java.net.UnknownHostException;
import saam.util.Array;

/**
 * This class represents a packet suitable for
 * transport within the SAAM architecture.  A SAAMPacket
 * contains a SAAMHeader and a payload and methods for accessing
 * both fields.
 */
public class SAAMPacket {

  /**
   * The header portion of this packet.
   */
  private SAAMHeader header;

  /**
   * The data portion of this packet.
   */
  private byte[] payload;

  /**
   * The byte array representation of this packet.
   */
  private byte[] packet;

      /**
   * Constructs a SAAM packet given the fields to be
   * associated with this packet.
   * @param header The SAAMHeader.
   * @param payload A byte array representing the payload.
      */
     public SAAMPacket(SAAMHeader header, byte[] payload){

    this.header = header;
    this.payload = payload;
    packet = Array.concat(packet,header.getHeader());
    packet = Array.concat(packet,payload);
      }

  /**
   * Constructs a SAAM packet given a byte array
   * containing the properly formatted fields to be
   * associated with this packet.
   * @param packet The byte representation of a SAAM packet.
   */
  public SAAMPacket(byte[] packet) throws UnknownHostException{

    this.packet = packet;
    header = new SAAMHeader(Array.getSubArray(
      packet,0,SAAMHeader.length));
    payload = Array.getSubArray(
      packet,SAAMHeader.length,packet.length);
  }

  /**
   * Returns the SAAM header.
   * @return The SAAM header as a SAAMHeader object.
   */
```

```
  public SAAMHeader getHeader(){
    return header;
  }

  /**
   * Returns the SAAM packet payload as a byte array.
   * @return The SAAM packet payload as a byte array.
   */
  public byte[] getPayload(){
    return payload;
  }
  /**
   * Returns the SAAM packet as a byte array.
   * @return The SAAM packet as a byte array.
   */
  public byte[] getBytes(){
    return packet;
  }
  public int length(){
    try{
      return packet.length;
    }catch(NullPointerException npe){
      return 0;
    }
  }

}
```

```java
package saam.net;

import saam.util.Array;
import saam.util.PrimitiveConversions;

/**
 * This class constructs a standard UDP header.
 * An UDP Header is an element within every packet that flows
 * through the SAAM architecture.  Instantiation of this object
 * constructs a 40-byte array that can be used for transport.
 * The class contains several methods for access each element of
 * the IPv6 header.
 */
public class UDPHeader {

  /**
   * The number of bytes in an UDPHeader.
   */
  public static final byte length = 8;

      private short sourcePort=0;
      private short destPort=0;
      private short payloadLength=0;
      private short checksum=0;

      private byte[]          header = new byte[length];

  /**
   * This constructor is used to construct an UDPHeader given
   * an 8-byte array containing properly ordered fields of a
   * standard UDP header.
   */
  public UDPHeader(byte[] data){
    this.header=data;
    sourcePort = PrimitiveConversions.getShort(
      Array.getSubArray(data,0,2));
    destPort = PrimitiveConversions.getShort(
      Array.getSubArray(data,2,4));
    payloadLength = PrimitiveConversions.getShort(
      Array.getSubArray(data,4,6));
    checksum = PrimitiveConversions.getShort(
      Array.getSubArray(data,6,8));
  }

  /**
   * This constructor is used to construct an UDPHeader given
   * the values to be contained within the header.
   */
  public UDPHeader(short sourcePort, short destPort,
                   short payloadLength, short checksum){

    this.sourcePort = sourcePort;
    this.destPort = destPort;
    this.payloadLength = payloadLength;
    this.checksum = checksum;

    header = Array.concat(PrimitiveConversions.getBytes(sourcePort),
      PrimitiveConversions.getBytes(destPort));
    header = Array.concat(header,
      PrimitiveConversions.getBytes(payloadLength));
    header = Array.concat(header,
```

```
        PrimitiveConversions.getBytes(checksum));
  }

        /**
         * Returns the 8-byte UDPHeader.
         * @return The UDPHeader as a byte array of 8 bytes.
         */
        public byte[] getHeader(){
              return header;
        }

  /**
   * Returns the source port.
   * @return the source port.
   */
  public short getSourcePort(){
    return sourcePort;
  }

  /**
   * Returns the destination port.
   * @return the destination port.
   */
  public short getDestPort(){
    return destPort;
  }

  /**
   * Returns the length.
   * @return the length.
   */
  public short getPayloadLength(){
    return payloadLength;
  }

  /**
   * Returns the checksum.
   * @return the checksum.
   */
  public short getChecksum(){
    return checksum;
  }

  /**
   * The String representation of the UDPHeader.
   * @return the String representation of the UDPHeader.
   */
  public String toString(){
      return
      "\nUDPHeader:"+
            "\nsource:   "+(sourcePort&0xffff)+
            "\ndest:     "+(destPort&0xffff)+
        "\nlength:   "+(payloadLength&0xffff)+
        "\nchecksum: "+(checksum&0xffff);
  }
}
```

APPENDIX G. RESIDENTAGENT PACKAGE SOURCE CODE

```java
package saam.residentagent;

import saam.event.SaamListener;
import saam.control.ControlExecutive;
import saam.message.*;

/**
 * A ResidentAgent is an Object that can be delivered accross a SAAM
 * network to a router, perform some type of processing or monitoring
 * on that router, and then be replaced, forward its state to another
 * router, or uninstall itself.
 */
public interface ResidentAgent extends SaamListener{

  /**
   * Within this method, an agent provides the necessary calls to the
   * ControlExecutive that perform all necessary registration.  For
   * example, if an agent wants to monitor a specific port on the
   * router, the agent would call the monitorPort method in the
   * ControlExecutive.
   * @param controlExec The ControlExecutive on the router this agent
   *        is being installed on.
   */
  public void install(ControlExecutive controlExec);

  /**
   * When an agent is being replaced, the ControlExecutive will remove
   * the old agent from all channels it is allowed to talk on and from
   * all channels it is monitoring.  The uninstall method is called by
   * the ControlExecutive upon replacement in order to allow the old
   * agent a chance to perform any other cleanup that might be
necessary,
   * such as disposing of a user interface, for instance.
   */
  public void uninstall();

  /**
   * When an agent is about to be replaced, the ControlExecutive calls
   * the transferState method, passing the old agent a copy of the new
   * agent.  The old agent should then call the receiveState method on
   * the new agent, and pass to the new agent any messages that should
   * be passed.
   * @param replacement The agent that is replacing the old agent.
   */
  public void transferState(ResidentAgent replacement);

  /**
   * This method is called one or more times by an agent that is about
   * to be replaced by this agent.  The purpose of this method is to
   * enable a state transfer from the old agent to the new agent.
   * @param message The message to be passed from the old agent to the
   *        new agent.
   */
  public void receiveState(Message message);

  /**
   * When a ResidentAgent requests a flow by calling the requestFlow
method
   * of the ControlExecutive, the agent expects to be assigned a flow
and
   * to be notified when that flow is assigned.  This method provides a
```

```
     * mechanism for notifying the ResidentAgent that a FlowResponse has
     * arrived.
     */
    public void receiveFlowResponse(FlowResponse flowResponse);

    /**
     * Some resident agents are accessed by Objects on the router.  This
     * method provides the means for communication between an Object on
     * the router and this ResidentAgent.
     * @param message The Message the Object sends to this ResidentAgent.
     * @return The Message this ResidentAgent sends back to the Object
     *         performing the query.
     */
    public Message query(Message message);
}
```

```
package saam.residentagent;

import saam.control.ControlExecutive;
import saam.message.*;

/**
 * Occasionally, Objects on the router need access to ResidentAgents
 * on that router.  By implementing this interface, those objects can
 * be sure that when a ResidentAgent is replaced, they receive the new
 * agent.  ResidentAgentCustomers must register with the
ControlExecutive
 * in order to receive agent updates.  When a replacement arrives, the
 * ControlExecutive calls the replaceAgent method on all customers of
 * that ResidentAgent.
 */
public interface ResidentAgentCustomer{

  /**
   * When the ControlExecutive receives an agent that is to replace
   * an existing agent, the ControlExecutive calls the replaceAgent
   * method on each of the ResidentAgentCustomers of that ResidentAgent.
   * @param replacement The new ResidentAgent.
   */
  public void replaceAgent(ResidentAgent replacement);

  /**
   * Returns an array that contains the class names of the resident
   * agents that this Object desires to be a ResidentAgentCustomer of.
   * @return An array that contains the class names of the resident
   * agents that this Object desires to be a ResidentAgentCustomer of.
   */
  public String[] getAgentTypes();
}
```

```java
package saam.residentagent.router;

import java.util.*;
import java.net.InetAddress;
import java.net.UnknownHostException;

import saam.control.*;
import saam.residentagent.*;
import saam.router.*;
import saam.event.*;
import saam.net.*;
import saam.message.*;
import saam.util.*;

/**
 * The ARPCache is a table for storing the MAC address associated
 * with the IPv6Address of the next hop.
 */
public class ARPCache extends Hashtable
  implements TableResidentAgent, MessageProcessor{


  /**
   * @serial
   */
  private TableGui gui;

  /**
   * @serial
   */
  private Vector names = new Vector();

  /**
   * @serial
   */
  private ControlExecutive controlExec;

  /**
   * @serial
   */
  private String[] messageTypes =
    {"saam.message.ARPCacheEntry"};

  /**
   * Constructs an ARPCache with
   */
  public ARPCache(){
    super(1711, 0.5f);
  }

  public void install(ControlExecutive controlExec){
    names.add("Next Hop");
    names.add("Next MAC");
    int[] columnWidths = {250,50};
    gui=new TableGui(toString(), names, columnWidths);
    this.controlExec=controlExec;
    controlExec.registerMessageProcessor(this);
  }
  public void uninstall(){
    clear();
  }
```

```
public Message query(Message message){
    ARPCacheEntry entry = get((ARPCacheEntry)message);
  return (Message)entry;
}
public void transferState(ResidentAgent replacement){
  for(Enumeration e = elements();
      e.hasMoreElements();){
     replacement.receiveState(
        (ARPCacheEntry)e.nextElement());
  }//for
}
public void receiveState(Message message){
  add((ARPCacheEntry)message);
}
public void receiveFlowResponse(FlowResponse flowResponse){
}
public void processMessage(Message message){
  try{
    String flowZeroNextHop =
      controlExec.getFlowZeroNextHop().toString();
    ARPCacheEntry entry = (ARPCacheEntry)message;
    if(entry.length()==IPv6Address.length+1){
      add(entry);
      gui.fillTable(getTable());
    }else{
      remove(entry);
      gui.fillTable(getTable());
    }
    if(!controlExec.getArpCacheStatus() &&
       !flowZeroNextHop.equals(IPv6Address.DEFAULT_HOST)){
      if(containsKey(flowZeroNextHop)){
        controlExec.updateCoreServiceStatus(this, true);
      }else{
        controlExec.updateCoreServiceStatus(this, false);
      }
    }
  }catch(Exception e){message = null;}
}
public String[] getMessageTypes(){
  return messageTypes;
}
public void receiveEvent(SaamEvent event){
}

/**
 * If a ARPCacheEntry has already been constructed,
 * this method allows it to be entered into the table.
 * @param entry The ARPCacheEntry to be entered.
 */
private void add(ARPCacheEntry entry){
   put(entry.getNextHop().toString(), entry);
}

    /**
     * Removes an entry from the <em>ARP cache</em>.
 * @param nextHop The nextHop as an IPv6Address object.
 *                Used as the lookup key in the table.
     */
    private void remove(ARPCacheEntry entry){
    remove(entry.getNextHop().toString());
    }
```

```
  /**
   * Returns the entire contents of this ARPCache or null
   * if this ARPCache is empty.
   * @return A Vector of all entries currently
   *         in the ARP cache.
   */
  public Vector getTable(){
    if(isEmpty()) return null;
    Vector table = new Vector(size());

    for(Enumeration e = elements();
        e.hasMoreElements();){
       Vector oneRow = new Vector();
       ARPCacheEntry entry = (ARPCacheEntry)e.nextElement();
       oneRow.add(entry.getNextHop().toString());
       oneRow.add(""+entry.getNextMAC());
       table.add(oneRow);
    }//for
    return table;
  }//getEmulationTable()

      /**
   * Retrieves an entry from the <em>ARP cache</em>
   * without removing it.
   * @param nextHop The IPv6Address of the next hop. This
   *                value is used as the lookup key.
   * @return One record of the ARP Cache represented as
   *         an object.
      */
  private ARPCacheEntry get(ARPCacheEntry entry){
    IPv6Address nextHop = entry.getNextHop();
    return (ARPCacheEntry) get(nextHop.toString());
  }

  /**
   * Returns the contents of the <em>emulation table</em>
   * in the form of a String (useful for displaying the table).
   * @return A String representation of the contents of the
   *         entire table.
   */
  public String toString() {
    return "ARPCache";
  }//toString()
}
```

```java
package saam.residentagent.router;

import java.util.*;
import java.net.InetAddress;
import java.net.UnknownHostException;

import saam.control.*;
import saam.residentagent.*;
import saam.router.*;
import saam.net.*;
import saam.event.*;
import saam.message.*;
import saam.util.*;

/**
 * The <em>flow routing table</em> stores the service level
 * and nextHop (IPv6Address) associated with a given flowID.
 */
public class FlowRoutingTable extends Hashtable
  implements TableResidentAgent, MessageProcessor{


  /**
   * @serial
   */
  private TableGui gui;

  /**
   * @serial
   */
  private Vector names = new Vector();

  /**
   * @serial
   */
  private ControlExecutive controlExec;

  /**
   * @serial
   */
  private String[] messageTypes =
    {"saam.message.FlowRoutingTableEntry"};

  /**
   * @serial
   */
  private boolean FRT_UP   = true;

  /**
   * @serial
   */
  private boolean FRT_DOWN = false;

  /**
   * Constructs a flowRoutingTable with initial capacity of
   * capacity.  Initial capacity should be a prime number to
   * help distribute the entries evenly among the buckets.
   */
  public FlowRoutingTable(){
    super(1711, 0.5f);
```

```
  }

  public void install(ControlExecutive controlExec){
    names.add("Flow ID");
    names.add("SL");
    names.add("Next Hop");
    int[] columnWidths = {100,50,250};
    gui=new TableGui(toString(), names, columnWidths);
    this.controlExec=controlExec;
    controlExec.registerMessageProcessor(this);
  }
  public void uninstall(){
    clear();
  }
  public Message query(Message message){
    int flowID = ((FlowRoutingTableEntry)message).getFlowID();
    FlowRoutingTableEntry result = (FlowRoutingTableEntry)
      get(new Integer(flowID));
    return result;
  }
  /**
   * Returns the entire contents of this FlowRoutingTable or null
   * if this FlowRoutingTable is empty.
   * @return A Vector of all entries currently
   *         in the flow routing table.
   */
  public Vector getTable(){
    if(isEmpty()) return null;
    Vector table = new Vector(size());

    for(Enumeration e = elements();
        e.hasMoreElements();){
      Vector oneRow = new Vector();
      FlowRoutingTableEntry entry =
(FlowRoutingTableEntry)e.nextElement();
      oneRow.add(""+entry.getFlowID());
      oneRow.add(""+entry.getSL());
      oneRow.add(entry.getNextHop().toString());
      table.add(oneRow);
    }//for
    return table;
  }//getEmulationTable()

  public void transferState(ResidentAgent replacement){
    for(Enumeration e = elements();
        e.hasMoreElements();){
      replacement.receiveState(
        (FlowRoutingTableEntry)e.nextElement());
    }//for
  }
  public void receiveState(Message message){
    add((FlowRoutingTableEntry)message);
  }
  public void receiveFlowResponse(FlowResponse flowResponse){
  }
  public void processMessage(Message message){
    FlowRoutingTableEntry entry =
      (FlowRoutingTableEntry)message;
    if(entry.length()==3){
      remove(entry);
      gui.fillTable(getTable());
```

```
        if(entry.getFlowID()==0){
          controlExec.updateCoreServiceStatus(this,
            new IPv6Address());
        }
      }else{
        add(entry);
        gui.fillTable(getTable());
        if(entry.getFlowID()==0){
          controlExec.updateCoreServiceStatus(this,
            entry.getNextHop());
        }
      }

  }
  public String[] getMessageTypes(){
    return messageTypes;
  }
  public void receiveEvent(SaamEvent event){
  }

  /**
   * If a FlowRoutingTableEntry has already been constructed,
   * this method allows it to be entered into the table.
   * @param entry The FlowRoutingTableEntry to be entered.
   */
  private void add(FlowRoutingTableEntry entry){
    put((new Integer(entry.getFlowID())),entry);
  }

      /**
       * Removes an entry from the <em>flow routing table</em>.
   * @param flowID The flow ID represented as an integer.
       */
      private void remove(FlowRoutingTableEntry entry){
    remove(new Integer(entry.getFlowID()));
      }

  /**
   * Returns the contents of the <em>emulation table</em>
   * in the form of a String (useful for displaying the table).
   * @return A String representation of the contents of the
   *         entire table.
   */
  public String toString() {
    return "FlowRoutingTable";
  }//toString()
}
```

```
package saam.residentagent.router;

import java.net.*;
import java.util.Vector;

import saam.residentagent.*;
import saam.router.*;
import saam.event.*;
import saam.util.SAAMRouterGui;
import saam.control.*;
import saam.net.*;
import saam.message.*;

public class LsaGenerator extends Thread
  implements ResidentAgent, MessageProcessor{


  /**
   * @serial
   */
  private SAAMRouterGui gui;
  /**
   * @serial
   */
  private ControlExecutive controlExec;
  /**
   * @serial
   */
  private String myAddress;
  /**
   * @serial
   */
  private IPv6Address destHost = new IPv6Address();
  /**
   * @serial
   */
  private int assignedFlowID = 0;
  /**
   * @serial
   */
  private int packetsReceived, packetsSent;
  /**
   * @serial
   */
  private short sourcePort,destPort;
  private long[] delayServiceLevelZero = new long[10];
  private Vector inChannel = new Vector(4);
  private Vector outChannel = new Vector(4);

  /**
   * @serial
   */
  private String[] messageTypes =
    {};


  public LsaGenerator(){
    gui = new SAAMRouterGui("LsaGenerator");
  }//ServerProbe()

  public String[] getMessageTypes(){
```

```
    return messageTypes;
  }

  public void install(ControlExecutive controlExec){
    start();
  }
  public void run(){
    this.controlExec=controlExec;
    controlExec.registerMessageProcessor(this);
    try{
      myAddress = InetAddress.getLocalHost().getHostAddress();
      gui.setTextField("I'm on
"+InetAddress.getLocalHost().getHostName());
    }catch(UnknownHostException uhe){
      gui.sendText("I don't know who I am! "+uhe.toString());
    }
    sourcePort = (short)controlExec.listenToRandomPort(this);
    destPort = (short)controlExec.SAAM_CONTROL_PORT;
    try{
      for(int i=0;i<controlExec.getNumberOfInterfaces();i++){
        int channel = ProtocolStackEvent.
          FROM_INTERFACE_TO_SLQUEUE_START_CHANNEL+i;
        inChannel.add(new Integer(channel));
        controlExec.addListenerToChannel(this,channel);
        gui.sendText("Monitoring Channel: "+channel);
        channel = ProtocolStackEvent.
          FROM_SLQUEUE_TO_SCHEDULER_START_CHANNEL+i;
        outChannel.add(new Integer(channel));
        controlExec.addListenerToChannel(this,channel);
        gui.sendText("Monitoring Channel: "+channel);
      }
    }catch(ChannelException ce){
      gui.sendText(ce.toString());
    }
  }
  public void transferState(ResidentAgent replacement){
    gui.sendText("I'm being replaced...");
  }
  public void receiveState(Message message){
    gui.sendText("Replacing old agent");
  }
  public void receiveFlowResponse(FlowResponse flowResponse){
    //this agent is allowed by the ControlExecutive to talk
    //on flow zero, so no FlowRequest needs to be submitted.
  }
  private void sendLSA(LinkStateAdvertisement lsa){
    packetsSent++;
    IPv6Header ipv6Header = new IPv6Header(assignedFlowID,destHost);
    byte[] payload = (new String("Message "+(packetsSent))).getBytes();
    UDPHeader udpHeader = new UDPHeader(sourcePort,destPort,
      (short)payload.length,(short)0);
    IPv6Packet packet = new IPv6Packet(ipv6Header,udpHeader,payload);
    try{
      controlExec.send(this,lsa,assignedFlowID,sourcePort,
        controlExec.getServerIP(),destPort);
      gui.sendText(new String(payload)+" sent");
      gui.sendText(udpHeader.toString());
    }catch(FlowException fe){
      gui.sendText(fe.toString());
    }
  }
```

```java
  public void uninstall(){
  }
  public Message query(Message message){
    return message;
  }
  public void processMessage(Message message){
    gui.sendText("Received Message: "+message.toString());
    //here the server has requested that an LSA be sent.
    //Based on the parameters contained in the LsaRequest,
    //we would construct an LinkStateAdvertisement and
    //send it to the server
    //LinkStateAdvertisement lsa =
      //new LinkStateAdvertisement([parameters]);
    //sendLSA(lsa);
  }
  public void receiveEvent(SaamEvent se){
    ProtocolStackEvent pse = (ProtocolStackEvent)se;
    if(inChannel.contains(new Integer(se.getChannel_ID()))){
      gui.sendText("Packet going in.. ");
      if(pse.getServiceLevel()==0){
        packetsReceived++;
      }
    }else if(outChannel.contains(new Integer(se.getChannel_ID()))){
      gui.sendText("Packet just came out.. ");
    }
    if(packetsReceived%10==0){
      gui.sendText("Received "+packetsReceived+" packets.. creating
LSA");
      LinkStateAdvertisement lsa = new LinkStateAdvertisement((byte)0);
    }
    gui.sendText("An LSA Was Just Sent");
    //hard code an LSA for now
//    IPv6Address source = IPv6Address
//    LinkStateAdvertisement lsa =
//      new LinkStateAdvertisement();
  }//receiveEvent()

  public String toString(){
    return ("LsaGenerator");
  }
}
```

```java
package saam.residentagent.router;

import java.net.UnknownHostException;
import java.util.*;

import saam.*;
import saam.control.*;
import saam.router.*;
import saam.residentagent.*;
import saam.message.*;
import saam.util.*;
import saam.event.*;
import saam.net.*;

/**
 * This Scheduler looks in the service level queues of the Interface to
 * which it is connected and extracts packets based on a round-robin
 * algorithm.  Dequeued packets are forwarded to the outbound
 * NetworkInterfaceCard.
 */
public class Scheduler
   implements SaamTalker, Runnable,
     ResidentAgent{

  /**
   * The number of bytes in a SAAM network address.  This
   * variable is used in the routePacket() method to determine
   * the outbound interface when ARPing.
   */
  private static final int bytesToCheck = 5;
  private static int instanceNumber = 0;

  //for the thread
  private boolean started;
  private int interfaceNumber;
  private int populatedQueues;
  private int outBoundChannel;
  private int inBoundChannel;
  private long[] packetCounter =
    new long[Interface.numberOfServiceLevels];

  private IPv6Packet dataPacket;
  private SAAMRouterGui gui;
  private ControlExecutive controlExec;
  private Interface outboundInterface;

  private Object theLock= new Object();


  public void install(ControlExecutive controlExec){
    this.controlExec = controlExec;

    interfaceNumber = instanceNumber;
    try{
      this.outboundInterface =
        controlExec.getInterface(interfaceNumber);
    }catch(ArrayIndexOutOfBoundsException aioobe){
      //tried to access an interface with a number that
      //was too high.. so start the count over
      interfaceNumber = 0;
      instanceNumber = 0;
```

```
      started = true;
      this.outboundInterface =
        controlExec.getInterface(interfaceNumber);
    }

    gui = new SAAMRouterGui("Scheduler #"+interfaceNumber);
    gui.sendText("Installing...");
    instanceNumber++;

    int channel_ID = ProtocolStackEvent.
      FROM_SLQUEUE_TO_SCHEDULER_START_CHANNEL+interfaceNumber;
    try{
       controlExec.addTalkerToChannel(this,
         channel_ID);
       gui.sendText("Talking enabled on channel: " + channel_ID);
    }catch(ChannelException ce){
     gui.sendText(ce.toString());
    }

    outBoundChannel = ProtocolStackEvent.
            getFromSchedulerToNICChannel(interfaceNumber);
    try{
       controlExec.addTalkerToChannel(this,
         outBoundChannel);
       gui.sendText("Talking enabled on channel: " + outBoundChannel);
    }catch(ChannelException ce){
     gui.sendText(ce.toString());
    }
    inBoundChannel =
      ProtocolStackEvent.getFromInterfaceToSLQueueChannel(
        interfaceNumber);
    try{
       controlExec.addListenerToChannel(this, inBoundChannel);
       gui.sendText("Listening to channel: "+inBoundChannel);
    }catch(ChannelException ce){
      gui.sendText(ce.toString());
    }//try-catch

    gui.setTextField("I'm alive, but sleeping...");

    Thread schedulerThread = new Thread(this,
      "Scheduler"+System.currentTimeMillis());
    schedulerThread.start();

  }
  public void uninstall(){
  }
  public void transferState(ResidentAgent replacement){}
  public void receiveState(Message message){}
  public void receiveFlowResponse(FlowResponse flowResponse){
  }
  public Message query(Message message){
    //stubbed out
    return message;
  }

  /**
   * Here the service level queues are visited and packets are
   * dequeued round-robin, then sent to the forwardPacket method.
   */
     public void run(){
```

```
   gui.sendText("Thread started: "+ Thread.currentThread().getName());
        while(true){
       gui.setTextField("Looking in outbound SL queue's...");
       searchOutboundQueues();
       gui.setTextField(
         "All SL queues empty, I'm going to sleep");
         try{
             synchronized(theLock){
               gui.sendText("Waiting...");
               while(!started)  theLock.wait();
               gui.sendText("Resumed");
             }
         }catch(InterruptedException ie){
           gui.sendText(ie.toString());
         }//try-catch
        }//while(started)
     }//run()

 private synchronized void searchOutboundQueues() {
   int slCount = Interface.numberOfServiceLevels;
   while(populatedQueues>0){
     for(int i=0; i<slCount; i++){
       int queueID =
         Interface.SERVICE_LEVEL_QUEUE_START_INDEX + i;
       if (!(outboundInterface.isEmpty(queueID))){
         packetCounter[queueID]++;
         gui.sendText("dequeuing packet #"+packetCounter[queueID]+
           " from SL["+queueID+"]");
         forwardPacket(i,outboundInterface.getPacket(queueID));
         if (outboundInterface.isEmpty(queueID)){
           populatedQueues--;
         }//end if
       }//end if
     }//for
   }//while
   started=false;
 }//searchOutboundQueues()
 /**
  * Receives and processes the SaamEvent notification
  * generated by the packet talker to whom this Translator is
  * listening.
  * @param pe The SaamEvent to be processed.
  */
  public  void receiveEvent(SaamEvent se){
    gui.setTextField("Populated Service Levels: "+populatedQueues);
    ProtocolStackEvent pse = (ProtocolStackEvent)se;
    int channel = pse.getChannel_ID();
    int sl = pse.getServiceLevel();
    int numP = outboundInterface.getPacketCount(
      Interface.SERVICE_LEVEL_QUEUE_START_INDEX+sl);
    if (numP == 1){
         // if more than one arrives at same time, will miss!!
         populatedQueues++;
    }

    //this method doesn't actually use the SaamEvent
    if(!started){
       gui.sendText("packet arrived: channel:"+
         channel+" SL:"+sl);
       synchronized(theLock){
         started = true;
```

```
        theLock.notify();
      }
    }else{
      gui.sendText("Received event, already active");
      //future work..
    //perform an interrupt if the sl the arriving packet
    //is placed into is of higher priority than the service
    //level currently being polled.
    }//if-else
  }//receiveEvent()

 /**
  * Forwards the packet that was just dequeued from a service level
  * queue to the outbound NetworkInterfaceCard.
  * @param sl The service level this packet was dequeued from.
  * @param packet the byte array representation of this packet.
  */
 private void forwardPacket(int sl, byte[] packet){

   //send an event to indicate that a packet has been dequeued
   //from a service level queue.
   int channel_ID = ProtocolStackEvent.
     FROM_SLQUEUE_TO_SCHEDULER_START_CHANNEL+
       interfaceNumber;
   ProtocolStackEvent event = new ProtocolStackEvent(
       toString(),
       this,
       channel_ID,
       packet,
       sl);
   try{
     controlExec.talk(event);
   }catch(ChannelException tde){
     gui.sendText(tde.toString());
   }

   //since the nextHop was added to the packet before
   //the packet was enqueued into the Service Level Queue,
   //we strip it off here.
   IPv6Address nextHop = null;
   try{
     nextHop = new IPv6Address(Array.getSubArray(
       packet, 0, IPv6Address.length));
   }catch(UnknownHostException uhe){
     gui.sendText(toString()+": "+uhe.toString());
   }
   gui.sendText("Next Hop:\n"+nextHop.toString());
   IPv6Packet v6Packet = null;
   try{
     v6Packet = new IPv6Packet(packet);
   }catch(UnknownHostException uhe){
     gui.sendText("Scheduler: "+uhe.toString());
   }
   gui.sendText("Forwarding packet to my NIC; Payload length = " +
     v6Packet.getPayload().length);
   event = new ProtocolStackEvent(
     toString(),
     this,
     outBoundChannel,
     packet,
     sl,
```

```
      nextHop);
    try{
      controlExec.talk(event);
    }catch(ChannelException tde){
      gui.sendText(tde.toString());
    }
  }//forwardPacket()

  /**
   * Returns the String representation of this Object.
   * @return The String representation of this Object.
   */
      public String toString(){
              return "Scheduler"+interfaceNumber;
      }
}
```

```java
package saam.residentagent.server;

import saam.control.*;
import saam.residentagent.*;
import saam.server.*;
import saam.event.*;
import saam.message.*;
import saam.util.*;

public class ServerAgent
  implements ResidentAgent, MessageProcessor{

  private SAAMRouterGui gui;
  private ControlExecutive controlExec;
  private Server myServer;
  private String[] messageTypes =
    {"saam.message.Hello",
     "saam.message.FlowRequest",
     "saam.message.LinkStateAdvertisement"};

  public void install(ControlExecutive controlExec){
    gui=new SAAMRouterGui("ServerAgent");
    this.controlExec=controlExec;
    controlExec.registerMessageProcessor(this);
    myServer = new Server("classObject", controlExec);
//    mysServer = new Server("database", controlExec);
  }
  public void processMessage(Message message){
    try{
      if(message instanceof Hello){
        gui.sendText("Received Message: "+((Hello)message));
        gui.sendText("Calling Server method: processHello()");
        myServer.processHello((Hello)message);
      }else if(message instanceof FlowRequest){
        FlowRequest request = (FlowRequest)message;
        gui.sendText("Received Message: "+ request);
        gui.sendText(
          "Calling Server method: processFlowRequest()");
        myServer.processFlowRequest((FlowRequest)message);

      }else if(message instanceof LinkStateAdvertisement){
        gui.sendText("Received Message: "+
          ((LinkStateAdvertisement)message));
        gui.sendText("Calling Server method: processLSA()");
        myServer.processLSA(
          (LinkStateAdvertisement)message);
      }
    }catch(Exception e){message = null;}
  }
  public String[] getMessageTypes(){
    gui.sendText("Server queried my message types");
    gui.sendText("Sending: "+messageTypes[0]);
    return messageTypes;
  }
  public String toString() {
    String it = "ServerAgent listening for: ";
    for(int i=0;i<messageTypes.length;i++){
      it += "\n" + messageTypes[i];
    }
    return it;
  }//toString()
```

```
  //the following methods are stubbed out as they are not used.
  public void uninstall(){
  }
  public Message query(Message message){
    return message;
  }
  public void transferState(ResidentAgent replacement){
  }
  public void receiveState(Message message){
  }
  public void receiveFlowResponse(FlowResponse flowResponse){
  }
  public void receiveEvent(SaamEvent event){
  }
}
```

APPENDIX H. ROUTER PACKAGE SOURCE CODE

```java
package saam.router;

/**
 * FlowException are thrown for various reasons.  For
 * example, if an Object attempts to send traffic on a flow
 * that it does not own, a FlowException will be thrown.
 */
public class FlowException extends Exception{

  /**
   * Constructs a generic FlowException.
   */
  public FlowException() {
    super();
  }

  /**
   * Constructs a FlowException with a descriptive String.
   * @param s A String describing the Exception in more detail.
   */
  public FlowException(String s) {
    super(s);
  }
}
```

```java
package saam.router;

import java.util.Vector;
import java.util.TooManyListenersException;
import java.net.UnknownHostException;

import saam.control.*;
import saam.util.SAAMRouterGui;
import saam.util.Queue;
import saam.util.Array;
import saam.event.*;
import saam.net.IPv6Packet;
import saam.net.SAAMPacket;
import saam.message.InterfaceID;

/**
 * A router Interface contains a <em>network interface card</em>, one
 * queue for inbound traffic, and several service level queues for
 * outbound traffic.
 */

public class Interface implements SaamTalker, SaamListener{

  public static final int numberOfInboundQueues = 1;
  public static final int numberOfServiceLevels = 4;
  public static final int numberOfQueuesOnThisInterface =
    numberOfInboundQueues+numberOfServiceLevels;
  public static final int INBOUND_QUEUE = 0;
  public static final int SERVICE_LEVEL_QUEUE_START_INDEX =
    INBOUND_QUEUE + numberOfInboundQueues;

      private static byte instanceCounter=0;

      private byte interfaceInstance=0;
  private ControlExecutive controlExec;
  private int enqueueingInboundChannel;
  private int enqueueingOutboundChannel;

      private InterfaceID interfaceID;
      private NetworkInterfaceCard NIC;
  /**
   * A Vector containing all of the queues in this interface.
   */
  private Vector listOfQueues =
    new Vector(numberOfQueuesOnThisInterface);
      private SAAMRouterGui gui;
  private IPv6Packet outboundPacket;
  private SAAMPacket sif;
  private long inboundPacketCount, outboundPacketCount;

  /**
  * Constructs a new SAAMRouterInterface with the given interfaceID.
  * @param router An instance of the SAAMRouter that instantiated this
   * interface.  this is to enable the interface to register as a
listener
  * with the router.
  * @param interfaceID The numerical representation of this interface.
  * @param translator An instance of the Translator object instantiated
  * by the SAAMRouter.
  */
  public Interface(ControlExecutive controlExec,
```

```
  InterfaceID interfaceID){

   interfaceInstance=instanceCounter;
   instanceCounter++;
   this.controlExec = controlExec;
   this.interfaceID=interfaceID;
   gui = new SAAMRouterGui("Interface #"+
     interfaceInstance);
   gui.setTextFieldFontSize(10);

   //emptyQ() is a static method
   listOfQueues.add(INBOUND_QUEUE,Queue.emptyQ());
   for(int offset=0;offset<numberOfServiceLevels;offset++){
     listOfQueues.add(
       SERVICE_LEVEL_QUEUE_START_INDEX+offset,Queue.emptyQ());
   }

   //************************************************
   //**Add yourself the channels you wish to talk on***
   //************************************************
   enqueueingInboundChannel =
       ProtocolStackEvent.getEnqueuingInboundPacketChannel(
         interfaceInstance);
   addTalkerToChannel(enqueueingInboundChannel);
   //************************************************
   enqueueingOutboundChannel =
     ProtocolStackEvent.getFromInterfaceToSLQueueChannel(
       interfaceInstance);
   addTalkerToChannel(enqueueingOutboundChannel);
   //************************************************

   //Instantiates the NIC for this Interface
   NIC = new NetworkInterfaceCard(controlExec,
       interfaceID.getMAC(),interfaceInstance);

   //************************************************
   //****Monitor the channels you wish to listen on****
   //************************************************
   int channel = ProtocolStackEvent.
           getFromNICToInterfaceChannel(
               interfaceInstance);
   addListenerToChannel(channel);
   //************************************************
   channel = ProtocolStackEvent.
           getFromRoutingAlgorithmToInterfaceChannel(
               interfaceInstance);
   addListenerToChannel(channel);
   //************************************************

   gui.setTextField("My ID: " + interfaceID.toString());
   gui.sendText("I'm alive and listening");
 }//Interface()

 private void addTalkerToChannel(int channel_ID){
   try{
      controlExec.addTalkerToChannel(this,
        channel_ID);
   }catch(ChannelException ce){
    gui.sendText(ce.toString());
   }
 }//addTalkerToChannel()
```

```
private void addListenerToChannel(int channel_ID){
  try{
     controlExec.addListenerToChannel(this, channel_ID);
     gui.sendText("Listening to channel: "+channel_ID);
  }catch(ChannelException ce){
    gui.sendText(ce.toString());
  }//try-catch
}

/**
 * Gets the IPv6 address for this interface.
 * @return The IPv6 address for this interface.
 */
public InterfaceID getID(){
  return interfaceID;
}


/**
 * Returns the number of packets in the queue
 * of the inbound interface.
 * @return     The (int) number of packets
 * @see        saam.util.Queue#getCount()
 */
public synchronized int getPacketCount(int queueID){
  return ((Queue)listOfQueues.get(queueID)).getCount();
}

/**
 * Determines if the queue of the inbound interface
 * is empty.
 * @return     true if the queue is empty
 * @see        saam.util.Queue#isemptyQ()
 */
public synchronized boolean isEmpty(int queueID){
  return ((Queue)listOfQueues.get(queueID)).isemptyQ();
}

/**
 * Returns the packet at the front of the queue of the
 * inbound interface.
 * @return     A single packet (byte[]) from the front of the queue
 */
public synchronized byte[] getPacket(int queueID){
 Queue q = (Queue)listOfQueues.get(queueID);
  //retrieve the packet from the queue
  byte[] packet = (byte[])q.peek();
  //remove the packet from the queue
  q = q.dequeue();
  return packet;
}

/**
 * Return a reference to the packet at the front of the queue of the
 * inbound interface.
 * @return     A single packet (byte[]) from the front of the queue
 * Doesn't dequeue packet. for debug examination only.
 */
public synchronized byte[] peekPacket(int queueID){
 Queue q = (Queue)listOfQueues.get(queueID);
```

```
    //retrieve the packet from the queue
    byte[] packet = (byte[])q.peek();
    return packet;
  }


  /**
   * This method receives <code>ProtocolStackEvent</code> notifications
from
   * Talkers.  Actions are taken based on the source
   * of the <code>ProtocolStackEvent</code>.
   * @param pe The ProtocolStackEvent received.
   */
  public synchronized void receiveEvent(SaamEvent pe){

    ProtocolStackEvent pse = (ProtocolStackEvent)pe;
    int channel = pse.getChannel_ID();
    byte[] packet = pse.getPacket();

    if(channel==ProtocolStackEvent.getFromNICToInterfaceChannel(
      interfaceInstance)){
        IPv6Packet v6Packet = null;
        try{
          v6Packet = new IPv6Packet(packet);
        }catch(UnknownHostException uhe){
          gui.sendText("Couln't instantiate IPv6Packet");
        }
        //inbound packet.
        inboundPacketCount++;
        gui.sendText("Received Packet #"+inboundPacketCount+
          " on channel: "+channel);
        gui.sendText("Source: " +
v6Packet.getHeader().getSource().toString());
        gui.sendText("Came from: "+pse.getSource());
        gui.sendText("Packet length = " + packet.length +
          "; Payload length = " + v6Packet.getPayload().length +
          "; Flow label = " + v6Packet.getHeader().getFlowLabel());
        Queue inQ = (Queue)listOfQueues.get(INBOUND_QUEUE);
        inQ.enqueue(packet);
        gui.sendText("Inbound queue count: "+
          inQ.getCount()+".");

        ProtocolStackEvent event = new ProtocolStackEvent(
          toString(),
          this,
          enqueueingInboundChannel,
          packet);
        try{
          controlExec.talk(event);
        }catch(ChannelException tde){
          gui.sendText(tde.toString());
        }

    }else if(channel==ProtocolStackEvent.
        getFromRoutingAlgorithmToInterfaceChannel(
          interfaceInstance)){

        IPv6Packet v6Packet = null;
        try{
          // XXX need to be changed when MAC addition is moved to NIC
card
```

```
          v6Packet = new
IPv6Packet(Array.getSubArray(packet,1,packet.length));
        }catch(UnknownHostException uhe){
          gui.sendText("Couldn't instantiate IPv6Packet");
        }
        //outbound packet
        outboundPacketCount++;
        gui.sendText("Received outbound packet #"+outboundPacketCount+
          " on channel: "+channel);
        gui.sendText("Packet length = " + packet.length +
          "; Payload length = " + v6Packet.getPayload().length);
        gui.sendText("Source: "+v6Packet.getHeader().getSource());
        gui.sendText("Dest   : "+v6Packet.getHeader().getDest());
        int sl = pse.getServiceLevel();
        int outChannel =
          ProtocolStackEvent.getFromInterfaceToSLQueueChannel
            (interfaceInstance);
        Queue outQ = (Queue)listOfQueues.get(
          SERVICE_LEVEL_QUEUE_START_INDEX+sl);

        //A hack: insert nextHop onto the front of the packet
        //so it can be retreived from the scheduler when
        //dequeued.  This way we ensure that the scheduler
        //associates this packet with the appropriate service
        //level.
        packet = Array.concat(pse.getNextHop().getAddress(),
          packet);

        outQ.enqueue(packet);
        gui.sendText("service level["+sl+"] count: "+
          outQ.getCount());
        gui.sendText("Next Hop: "+pse.getNextHop().toString());
        gui.sendText("talking on channel: "+outChannel);

        ProtocolStackEvent event = new ProtocolStackEvent(
          toString(),
          this,
          outChannel,
          packet,
          sl,
          pse.getNextHop());
        try{
          controlExec.talk(event);
        }catch(ChannelException tde){
          gui.sendText(tde.toString());
        }

    }//if-else
  }//packetReceived
  public boolean equals(Interface in){
    return this.interfaceID.equals(in.getID());
  }
  /**
   * Returns a <code>String</code> representation of this object
   * @return The <code>String</code> representation of this object
   */
  public String toString(){
     return "Interface"+interfaceInstance;
  }
}
```

```
package saam.router;

import java.net.*;
import java.util.*;
import java.io.*;

import saam.control.*;
import saam.event.*;
import saam.net.*;
import saam.util.Array;

/**
 * Used to send/receive <em>IPv6</em> Packets over an
 * IPv6-capable wire (The <code>Translator</code> <em>
 * emulates</em> the IPv6-capable wire in this case.
 */
public class NetworkInterfaceCard
  implements SaamTalker, SaamListener{

      private ControlExecutive controlExec;
  /**
   * The MAC address assigned to this <code>
   * NetworkInterfaceCard</code>.
   */
  private byte MAC;
  private int thisInstance;

  /**
   * Constructs a NIC card that will listen to events from
   * <em>translator</em>.
   * @param MAC The MAC address to be assigned to this NIC card
   * @param translator The <em>translator</em> that will talk
   *                   to this NIC card.
   */
      public NetworkInterfaceCard(ControlExecutive controlExec,
    byte MAC, int thisInstance){

    this.thisInstance=thisInstance;
    this.controlExec=controlExec;
            this.MAC=MAC;

//*******************************************************/
// Set up talking mechanism
//*******************************************************/
    //acquire the channels you wish to talk on
    int channel = ProtocolStackEvent.
            getFromNICToInterfaceChannel(thisInstance);
    try{
      controlExec.addTalkerToChannel(this,channel);
    }catch(ChannelException ciue){
      System.out.println(ciue.toString());
    }

    channel = ProtocolStackEvent.
            FROM_NICS_TO_TRANSLATOR_CHANNEL;
    try{
      controlExec.addTalkerToChannel(this,channel);
    }catch(ChannelException ce){
      System.out.println(ce.toString());
    }
```

```
//****************************************************/
//Enable listening
//****************************************************/
    channel = ProtocolStackEvent.FROM_TRANSLATOR_TO_NICS_CHANNEL;
    try{
      controlExec.addListenerToChannel(this,channel);
    }catch(ChannelException ce){
     System.out.println(ce.toString());
    }//try-catch

    channel = ProtocolStackEvent.getFromSchedulerToNICChannel(
      thisInstance);
    try{
      controlExec.addListenerToChannel(this,channel);
    }catch(ChannelException ce){
     System.out.println(ce.toString());
    }//try-catch
      }//NetworkInterfaceCard

  /**
   * Receives and processes the ProtocolStackEvent notification
   * that was sent on a Channel this NetworkInterfaceCard is listening
   * to.
   * @param pe The ProtocolStackEvent to be processed.
   */
  public void receiveEvent(SaamEvent se){
    ProtocolStackEvent pse = (ProtocolStackEvent)se;
    int channel = pse.getChannel_ID();
    byte[] packet = pse.getPacket();
    if(channel==
      ProtocolStackEvent.FROM_TRANSLATOR_TO_NICS_CHANNEL){

      int theMAC = packet[0];
      if(theMAC == MAC){
        ProtocolStackEvent event = new ProtocolStackEvent(
          toString(),
          this,
          ProtocolStackEvent.getFromNICToInterfaceChannel(
            thisInstance),
          stripMAC(packet));
          try{
            controlExec.talk(event);
          }catch(ChannelException tde){
            System.out.println(tde.toString());
          }
      }//if(theMAC == MAC)

    }else if(channel==ProtocolStackEvent.
        getFromSchedulerToNICChannel(thisInstance)) {

        //the NIC simply forwards the packet on to the
        //Translator... no added value here.  This should
        //be looked at.
        ProtocolStackEvent event = new ProtocolStackEvent(
          toString(),
          this,
          ProtocolStackEvent.FROM_NICS_TO_TRANSLATOR_CHANNEL,
          packet);
          try{
            controlExec.talk(event);
          }catch(ChannelException tde){
```

```
          System.out.println(tde.toString());
        }
    }//if-else
      }//packetReceived

  /**
   * Strips the MAC address field from packet.
   * @param packet The packet containing the MAC address
   *               and the IPv6Packet to be forwarded.
   * @return The IPv6Packet as a byte array.
   */
      public byte[] stripMAC(byte[] packet){
    return Array.getSubArray(packet,1,packet.length);
      }//stripMAC

  /**
   * Returns the String representation of this
   * NetworkInterfaceCard.
   * @return The String representation of this
   *         NetworkInterfaceCard.
   */
  public String toString(){
    return "NIC"+thisInstance;
  }
}
```

```
package saam.router;

import java.net.UnknownHostException;
import java.util.*;

import saam.*;
import saam.control.*;
import saam.residentagent.*;
import saam.residentagent.router.*;
import saam.util.*;
import saam.event.*;
import saam.net.*;
import saam.message.*;

/**
 * The RoutingAlgorithm looks in the inbound queues of all Interfaces on
 * this router and extracts packets based on a round-robin
 * algorithm.  Dequeued packets are looked into to determine the
destination.
 * If the packet is destined for any Interface on this router, the
packet
 * is forwarded to the application layer; otherwise the packet is
forwarded
 * to the outbound interface.
 */
public class RoutingAlgorithm
   implements ResidentAgentCustomer,
     SaamTalker, SaamListener, Runnable{

  /**
   * The number of bytes in a SAAM network address.  This
   * variable is used in the routePacket() method to determine
   * the outbound interface when ARPing.
   */
  private static final int bytesToCheck = 5;

  /**
   * The agentTypes the RoutingAlgorithm registers to process.
   */
  private static final String[] agentTypes =
    {"saam.residentagent.router.ARPCache",
     "saam.residentagent.router.FlowRoutingTable"};

  private ResidentAgent arpCache, flowRoutingTable;
  private ControlExecutive controlExec;
      private SAAMRouterGui gui =
    new SAAMRouterGui("RoutingAlgorithm");
      private Vector interfaces = new Vector();
  private int interfaceCount;
  private int populatedQueues;
  private Interface outboundInterface;
//  private boolean started = true;
  private boolean started;
  private long packetCounter;

  private Thread owner; //not currently used!!!!
  private Object theLock= new Object();

  /**
   * The ControlExecutive instantiates the RoutingAlgorithm, passing it
   * a copy of the ControlExecutive so the RoutingAlgorithm can call
```

```
  * methods on that ControlExecutive.  The ControlExecutive
instantiates
  * the arpCache ResidentAgent and passes it to the RoutingAlgorithm.
  * @param controlExec The ControlExecutive this RoutingAlgorithm will
  * use to perform certain operations.
  * @param arpCache The ARPCache ResidentAgent this RoutingAlgorithm
will
  * use to determine the MAC address of the next hop for outbound
packets.
  */
     public RoutingAlgorithm(ControlExecutive controlExec,
   ResidentAgent arpCache){

   this.controlExec = controlExec;
   flowRoutingTable = new FlowRoutingTable();
   this.arpCache = arpCache;

controlExec.registerMessageProcessor((MessageProcessor)flowRoutingTable)
;
   controlExec.registerMessageProcessor((MessageProcessor)arpCache);
   controlExec.registerCustomer(this);
   //add talker
   int channel_ID = ProtocolStackEvent.
     FROM_ROUTINGALGORITHM_TO_TRANSPORTINTERFACE_CHANNEL;
   try{
      controlExec.addTalkerToChannel(this,channel_ID);
      gui.sendText("Talking enabled on channel: " + channel_ID);
   }catch(ChannelException ce){
    gui.sendText(ce.toString());
   }
   //add listener
   channel_ID = ProtocolStackEvent.
     FROM_TRANSPORTINTERFACE_TO_ROUTINGALGORITHM_CHANNEL;
   try{
      controlExec.addListenerToChannel(this, channel_ID);
      gui.sendText("Listening to channel: "+channel_ID);
   }catch(ChannelException ce){
     gui.sendText(ce.toString());
   }//try-catch

   Thread algorithmThread = new Thread(this,
     "Routing Algorithm");
   algorithmThread.start();
           gui.setTextField("I'm sleeping...");
     }//end RoutingAlgorithm()

 /**
  * The ControlExecutive calls this method to allow the
RoutingAlgorithm
  * to set up the means of communication between itself and this new
  * Interface.
  * @param nextInterface The Interface this RoutingAlgorithm will add
  * to the list of Interfaces it monitors.
  */
 public void addInterface(Interface nextInterface){
   //add talker
   gui.sendText("Adding "+nextInterface.toString());
   int channel_ID = ProtocolStackEvent.
     FROM_ROUTINGALGORITHM_TO_INTERFACE_START_CHANNEL+
       interfaceCount;
   try{
```

```
      controlExec.addTalkerToChannel(this, channel_ID);
      gui.sendText("Talking enabled on channel: " + channel_ID);
    }catch(ChannelException ce){
     gui.sendText(ce.toString());
    }
    //add listener
    channel_ID = ProtocolStackEvent.
      ENQUEUING_INBOUND_PACKET_START_CHANNEL+
        interfaceCount;
    try{
      controlExec.addListenerToChannel(this, channel_ID);
      gui.sendText("Listening to channel: "+channel_ID);
    }catch(ChannelException ce){
      gui.sendText(ce.toString());
    }//try-catch
    interfaces.add(nextInterface);
    gui.sendText("Monitoring "+interfaces.size()+
      " interfaces");
    gui.sendText("Waiting...");
    interfaceCount++;
  }//addInterface()

  /**
   * Returns an array that contains the class names of the resident
   * agents that this Object desires to be a ResidentAgentCustomer of.
   * @return An array that contains the class names of the resident
   * agents that this Object desires to be a ResidentAgentCustomer of.
   */
  public String[] getAgentTypes(){
    return agentTypes;
  }//getAgentTypes()

  /**
   * When the ControlExecutive receives an agent that is to replace
   * an existing agent, the ControlExecutive calls the replaceAgent
   * method on each of the ResidentAgentCustomers of that ResidentAgent.
   * @param agent The new ResidentAgent.
   */
  public synchronized void replaceAgent(
    ResidentAgent agent){
    if(agent.getClass().getName().equals(agentTypes[0])){
//        ARPCache arp = (ARPCache)agent;
      gui.sendText("ARPCache is being replaced...");
      this.arpCache=agent;
      gui.sendText("New ARPCache installed...");
      gui.sendText(arpCache.toString());
    }else
    if(agent.getClass().getName().equals(agentTypes[1])){
      gui.sendText("FlowRoutingTable is being replaced...");
      this.flowRoutingTable=agent;
      gui.sendText("New FlowRoutingTable installed...");
      gui.sendText(flowRoutingTable.toString());
    }else{
      gui.sendText("RoutingAlgorithm is not a customer "+
        "of "+agent.toString());
    }
  }//replaceAgent()
  /**
   * The algorithm for determining which inbound queue to select from
   * next is contained here.  For this version of the SAAM router, the
   * algorithm merely visits the queues in round-robin fashion.  When
```

```
   * a packet is dequeued, it is sent to the routeInboundPacket method.
   */
  public void run(){

          //round-robin scheduler

      gui.sendText("Monitoring "+interfaceCount+
        " interfaces");

          while(true){
        gui.sendText("Looking in inbound queue's...");
        searchInboundQueues();
        gui.sendText(
          "All inbound queues empty, I'm going to sleep");
        try{
          synchronized(theLock){
            gui.sendText("Waiting...");
            while(!started)
              theLock.wait();
            gui.sendText("Resumed");
          }
        }catch(InterruptedException ie){
          gui.sendText(ie.toString());
        }
         }//while(started)
      }//run()
 /**
  * Searches all inbound queues for a packet.  Each queue is searched
  * until all packets in the queue have been removed.  All queues are
  * repeatedly searched until two conditions are met:
  * 1) All queues are empty and
  * 2) No Interface has enqueued a packet during the search.
  */
  private void searchInboundQueues(){
    //Continue sweeping the queues and routing packets until all queues
    //are empty and no Interface sends a notification that a packet
    //is being enqueued.
    int queueID = Interface.INBOUND_QUEUE;
    gui.sendText("Searching... populatedQueues = "+populatedQueues);
    while ( populatedQueues > 0)
    {
      for (int i=0; i<interfaceCount; i++){
        Interface thisInterface = (Interface)interfaces.get(i);
        // round robin to achieve fairness
        if (!(thisInterface.isEmpty(queueID))){
          gui.sendText("Packet count: " +
thisInterface.getPacketCount(queueID));
          gui.sendText("dequeuing packet from "+
            thisInterface.toString());
          routeInboundPacket(thisInterface.getPacket(
            Interface.INBOUND_QUEUE));
          if ( thisInterface.isEmpty(queueID) )
            populatedQueues--;
        }//end if
      }//end for
    }//while

    started = false;
  }//searchInboundQueues()
```

```
    /**
     * Receives and processes SaamEvents received on the Channel that
     * has been designated as the communication Channel for traffic from
the
     * TransportInterface to the RoutingAlgorithm.
     * @param se The SaamEvent that contains the outbound packet.
     */
    public void receiveEvent(SaamEvent se){
     //this doesn't actually use the SaamEvent
      Interface thisInterface = (Interface)se.getTalker();
      int numP = thisInterface.getPacketCount(Interface.INBOUND_QUEUE);
      if (numP == 1){
           // if more than one arrives at same time, will miss!!
           populatedQueues++;
      }

      // Print out debugging information
      IPv6Packet packet = null;
      try{
        packet = new
IPv6Packet(thisInterface.peekPacket(Interface.INBOUND_QUEUE));
      }catch(UnknownHostException uhe){};
      gui.sendText("Payload length = " + packet.getPayload().length);
      gui.sendText("Source: "+
        packet.getHeader().getSource().toString());
      gui.sendText("Dest: "+
        packet.getHeader().getDest().toString());

      if(!started){
         gui.sendText("packet arrived from : " + thisInterface+". I'm
activating");
         synchronized(theLock){
           started = true;
           theLock.notify();
         }
      }else{
         gui.sendText("packet arrived from : " + thisInterface+". already
active");
         gui.sendText("Packet count for this queue = " + numP);
      }//if(!started)


    }//packetReceived

   /**
    * This method answers the question - "Is this packet destined for
    * an Interface on this router?".
    * @param dataPacket The packet to be tested.
    * @return True if the packet is destined for an Interface on this
router.
    */
   public boolean isApplicationLayerPacket(IPv6Packet dataPacket){
     for(int i=0;i<interfaces.size();i++){
       Interface thisInterface = (Interface)interfaces.get(i);
       if(dataPacket.getHeader().getDest().toString().equals(
         thisInterface.getID().getIPv6().toString()))
           return true;
     }
     return false;
   }//isApplicationLayerPacket()
```

```
  /**
   * The purpose of this method is to determine which Interface is
   * connected to the Interface this packet is travelling to next.
   * Compares the network portion of the nextHop provided with the
   * network portions of the IPv6Addresses of each of the the Interfaces
   * provided in the Vector.
   * @param interfaces The Vector of Interfaces on this router.
   * @param nextHop The IPv6Address to be compared.
   * @return The Interface that is connected to the Interface this
packet
   * is travelling to next.
   */
  public Interface determineOutboundInterface(Vector interfaces,
    IPv6Address nextHop){
    byte[] nextHopBytes = nextHop.getAddress();
    for(int i=0;i<interfaces.size();i++){
      Interface thisInterface = (Interface)interfaces.get(i);
      //cycle through all interfaces
      //checking network address against nextHop.
      int match = 0;
      byte[] outboundInterfaceBytes =
        thisInterface.getID().getIPv6().getAddress();
      for(int index=0;index<bytesToCheck;index++){
        if((nextHopBytes[index]&0xFF)==
          (outboundInterfaceBytes[index]&0xFF)){
          match++;
        }//if
      }//for
      if(match==bytesToCheck){
        return thisInterface;
      }//if
    }//for
    return null;
  }//determineOutboundInterface()

  /**
   * Looks into the packet to determine whether it should be sent to
   * the application layer on this router or forwarded on to the next
hop.
   * @param whichInterface The inbound Interface the packet was dequeued
   *        from.
   * @param packet The byte array that represents the packet.
   */
    public void routeInboundPacket(byte[] packet){
    packetCounter++;
    IPv6Packet dataPacket = null;
    try{
      dataPacket = new IPv6Packet(packet);
    }catch(UnknownHostException uhe){};
    gui.sendText("Packet#: "+packetCounter+", Size = " +
      packet.length + "; Payload length = " +
dataPacket.getPayload().length);
    gui.sendText("Source: "+
      dataPacket.getHeader().getSource().toString());
    gui.sendText("Dest: "+
      dataPacket.getHeader().getDest().toString());
    if(isApplicationLayerPacket(dataPacket)){
      gui.sendText("Application layer packet.");
      gui.sendText("Forwarding to Transport Interface.");
      ProtocolStackEvent event = new ProtocolStackEvent(
        toString(),
```

```
        this,
        ProtocolStackEvent.
          FROM_ROUTINGALGORITHM_TO_TRANSPORTINTERFACE_CHANNEL,
        dataPacket.getBytes());
        try{
          controlExec.talk(event);
        }catch(ChannelException tde){
          gui.sendText(tde.toString());
        }
            }else{
      forwardPacket(dataPacket);
            }
}//routeInboundPacket()

/**
 * Checks to see whether or not an entry is in the ARPCache.
 * @param entry The ARPCacheEntry to be verified.
 * @return True if the entry is in the ARPCache.
 */
public boolean checkARPCache(ARPCacheEntry entry){
    ARPCacheEntry resultEntry =
      (ARPCacheEntry)arpCache.query(entry);
    return resultEntry!=null;
}//checkARPCache()

/**
 * Forwards the outbound packet to the appropriate Interface
 * @param packet A byte array representation of the outbound packet.
 */
public void forwardPacket(IPv6Packet packet){
      IPv6Header v6Header = packet.getHeader();
      gui.sendText("Forwarding packet");
    int flowID = v6Header.getFlowLabel();
    gui.sendText("Flow id: "+flowID);

    //to determine the next hop, we consult the FlowRoutingTable by
    //calling its query method and passing a FlowRoutingTableEntry
    //containing the flowID.
    Message message = (Message)(new FlowRoutingTableEntry(flowID));

    //The FlowRoutingTableEntry that is returned here contains the
    //next hop and service level we are looking for.
    FlowRoutingTableEntry frtEntry = (FlowRoutingTableEntry)
          flowRoutingTable.query(message);
    try{
      IPv6Address nextHop = frtEntry.getNextHop();
      gui.sendText("next hop: "+nextHop);
      byte sl = frtEntry.getSL();
      gui.sendText("SL: "+sl);

      //Now ARP to determine the MAC address of the next hop.
      message = (Message)(new ARPCacheEntry(nextHop));
      ARPCacheEntry entry = (ARPCacheEntry)arpCache.query(message);

      try{
        byte nextMAC = entry.getNextMAC();
        gui.sendText("nextMAC: "+nextMAC);
        outboundInterface = determineOutboundInterface(
          interfaces, nextHop);
        gui.sendText("outboundInterface: "+outboundInterface);
```

```
if(v6Header.getSource().toString().equals(IPv6Address.DEFAULT_HOST)){
            gui.sendText("Setting Source");
            v6Header.setSource(outboundInterface.getID().getIPv6());
            packet.setHeader(v6Header);
          }

          gui.sendText("Source: " + v6Header.getSource());
          gui.sendText("Dest: " + v6Header.getDest());
          gui.sendText("Appending next hop MAC address " +
            (nextMAC&0xff));
          // XXX -- needs to be moved to NIC card
          byte[] outboundPacket =
Array.concat(nextMAC,packet.getBytes());
          gui.sendText("Forwarding packet to: "+
            outboundInterface);

          //send a SaamEvent to the appropriate outbound interface.
          //This SaamEvent contains the service level among other
things.
          ProtocolStackEvent event = new ProtocolStackEvent(
            toString(),
            this,
            ProtocolStackEvent.
              getFromRoutingAlgorithmToInterfaceChannel(
                interfaces.indexOf(outboundInterface)),
            outboundPacket,
            sl,
            nextHop);
          try{
            controlExec.talk(event);
          }catch(ChannelException tde){
            gui.sendText(tde.toString());
          }
        }catch(NullPointerException npe){
          gui.sendText("Next Hop: "+nextHop+" is not in the ARPCache");
          gui.sendText("Packet Dropped\n");
        }
      }catch(NullPointerException npe){
        gui.sendText("Flow "+flowID+" is not in the FlowRoutingTable");
        gui.sendText("Packet Dropped\n");
      }
      }//forwardPacket()

  /**
   * Returns a <code>String</code> representation of this object
   * @return The <code>String</code> representation of this object
   */
      public String toString(){
            return "Routing Algorithm";
      }//toString()
}
```

```java
package saam.router;

import java.net.*;
import java.util.*;
import java.io.*;

import saam.*;
import saam.message.*;
import saam.net.*;
import saam.util.SAAMRouterGui;
import saam.event.*;
import saam.util.Array;
import saam.control.*;
import saam.residentagent.ResidentAgent;
import saam.message.MessageProcessor;

/**
 * The TransportInterface is at the top of the protocol stack.  The
 * RoutingAlgorithm passes packets destined for an Interface on this
 * router up to the TransportInterface.  The TransportInterface then
 * strips of the UDPHeader, determines which port the packet is destined
 * for, and forwards the packet on that port.
 */
public class TransportInterface
  implements SaamTalker,SaamListener{

  private SAAMRouterGui gui =
    new SAAMRouterGui("Transport Interface");

  private ControlExecutive controlExec;
  private int outChannel = ProtocolStackEvent.
      FROM_TRANSPORTINTERFACE_TO_ROUTINGALGORITHM_CHANNEL;

  private int packetCount;

  /**
   * Constructs a TransportInterface that will listen to events from
   * the <em>RoutingAlgorithm</em>.
   * @param ra The MAC address to be assigned to this NIC card
   * @param translator The <em>translator</em> that will talk
   *                   to this NIC card.
   */
      public TransportInterface(ControlExecutive controlExec){

    //here the TransportInterface registers to listen
    //on the control channel of the protocol stack and
    //registers to speak on the control channel in the
    //application layer
    this.controlExec = controlExec;
    try{
       controlExec.addTalkerToChannel(this, outChannel);
       gui.sendText("Talking approved on "+ outChannel);
    }catch(ChannelException ce){
      gui.sendText(ce.toString());
    }

          int channel_ID = ProtocolStackEvent.PACKETFACTORY_CHANNEL;
    try{
       controlExec.addTalkerToChannel(this, channel_ID);
       gui.sendText("Talking approved on "+
         (channel_ID<=ControlExecutive.MAX_PORT? "port: ":"channel: ") +
```

```
          channel_ID);
      }catch(ChannelException ce){
        gui.sendText(ce.toString());
      }

      channel_ID = ProtocolStackEvent.
        FROM_ROUTINGALGORITHM_TO_TRANSPORTINTERFACE_CHANNEL;
      try{
         controlExec.addListenerToChannel(this, channel_ID);
         gui.sendText("Listening to "+
           (channel_ID<=ControlExecutive.MAX_PORT? "port: ":"channel: ") +
           channel_ID);
      }catch(ChannelException ce){
        gui.sendText(ce.toString());
      }//try-catch
        }//TransportInterface

    /**
     * Used by the ControlExecutive to send a packet.  In this method,
     * the UDPHeader is constructed and then a TransportInterfaceEvent
     * is sent on the Channel the RoutingAlogorithm is listening to.
     * @param sender The Object sending the message.
     * @param saamPacket The SAAMPacket that will be wrapped in an
     *        IPv6Packet.
     * @param message The subclass of saam.message.Message to be sent.
     * @param flowID The flow ID that has been assigned for traffic from
sender
     *        destined for destHost.
     * @param sourcePort The port on the local machine that sender is
listening on.
     * @param destHost The IPv6Address of the destination.
     * @param destPort The port to which destHost is listening.
     */
    public IPv6Packet buildIPv6Packet(Object sender, SAAMPacket
saamPacket,
      int flowID, short sourcePort, IPv6Address destHost,
      short destPort){

        IPv6Header ipv6Header = new IPv6Header(flowID,destHost);
        byte[] payload = saamPacket.getBytes();
        short payloadLength = (short)payload.length;
        //construct the UDP header
        UDPHeader udpHeader = new UDPHeader(sourcePort,
          destPort, payloadLength, (short)0);

        gui.sendText("Building IPv6Packet:");
        gui.sendText("Encapsulated SAAMHeader:\n"+
          saamPacket.getHeader());
        SAAMPacket newSaamPacket = null;

        return new IPv6Packet(ipv6Header,udpHeader,payload);
    }

    /**
     * Receives and processes the SaamEvent notification
     * generated by the packet talker to whom this TransportInterface
     * is listening.
     * @param pe The SaamEvent to be processed.
     */
    public void receiveEvent(SaamEvent se){
      if(se.getChannel_ID()==ProtocolStackEvent.
```

```
      FROM_ROUTINGALGORITHM_TO_TRANSPORTINTERFACE_CHANNEL){
      packetCount++;
      gui.sendText("Received Event #"+packetCount+
        " on channel: "+se.getChannel_ID());
      ProtocolStackEvent pse = (ProtocolStackEvent)se;

      processSaamEvent(pse);
    }
      }//packetReceived

  /**
   * Here, application layer packets are received from the
RoutingAlgorithm.
   * This method looks at the UDP port the packet is destined for and
does
   * one of three things:<p>
   * 1. If the packet is destined for the SAAM_CONTROL_PORT, this method
   *    forwards the SAAMPacket to the PacketFactory for parsing.
   * 2. If the packet is destined for a regular port and there is a
listener
   *    on that port, the packet is sent to that port.
   * 3. If the packet is destined for a regular port but there is no
   *    listener on that port, the packet is dropped.
   *
   * Currently, there are no checks to determine whether the packet is
   * destined for a port within the allowable range.
   * @param pse The ProtocolStackEvent that was sent on a Channel the
   * TransportInterface was listening on.
   */
  private void processSaamEvent(ProtocolStackEvent pse){
    IPv6Packet v6Packet = null;
    try{
      v6Packet = new IPv6Packet(pse.getPacket());
    }catch(UnknownHostException uhe){
      gui.sendText("in processSaamEvent(): "+uhe.toString());
    }
    byte[] v6Payload = v6Packet.getPayload();
    String source = v6Packet.getHeader().getSource().toString();
    gui.sendText("Source: "+source);
    UDPHeader udp = v6Packet.getUDPHeader();
    short port = udp.getDestPort();
    gui.sendText("Dest Port: "+port);
    if(port == ControlExecutive.SAAM_CONTROL_PORT){
        gui.sendText("Passing event to PacketFactory");
        pse.setTalker(this);
        pse.setChannel_ID(ProtocolStackEvent.PACKETFACTORY_CHANNEL);
        pse.setPacket(v6Payload);
        gui.sendText("PayloadLength: "+v6Payload.length);
        try{
          controlExec.talk(pse);
        }catch(ChannelException ce){
          gui.sendText(ce.toString());
        }
    }else{
      gui.sendText("Sending up to dest port");
      gui.sendText(udp.toString());
      if(controlExec.hasListener(port)){
        if(port == ControlExecutive.SAAM_CONTROL_PORT){
          gui.sendText("SIF received...");
          gui.sendText("Notifying PacketFactory");
          gui.sendText("Packet length: "+v6Payload.length);
```

```
          ProtocolStackEvent event = null;
          event = new ProtocolStackEvent(
            toString(),
            this,
            ProtocolStackEvent.PACKETFACTORY_CHANNEL,
            v6Payload);
          try{
            controlExec.talk(event);
          }catch(ChannelException ce){
            gui.sendText(ce.toString());
          }
        }else{
          gui.sendText("Notifying port "+port+" listeners.");
          gui.sendText("Packet length: "+v6Payload.length);
          ApplicationEvent event = null;
          event = new ApplicationEvent(
            toString(),
            this,
            port,
            v6Packet.getBytes());
          try{
            controlExec.talk(event);
          }catch(ChannelException ce){
            gui.sendText(ce.toString());
          }
        }
      }else{
        gui.sendText("Packet dropped. No listeners on port: "+port);
        //take actions to dispose of the event
      }
    }
  }

  /**
   * Returns the String representation of this
   * TransportInterface.
   * @return The String representation of this
   *         TransportInterface.
   */
  public String toString(){
    return "TransportInterface";
  }
}
```

APPENDIX I. SERVER PACKAGE SOURCE CODE

```
package saam.server;

import saam.net.*;
import saam.message.*;
import saam.util.*;
import java.net.*;
import java.sql.*;
import java.util.*;
import java.io.*;

/**
 * The <em>ClassObjectStructure</em> is a Path Information Base object
within the
 * SAAM architecture that performs operations on class objects
containing the
 * information needed to obtain a picture of the network for use in
assigning
 * flows to paths.
 */
public class ClassObjectStructure extends PathInformationBase{

/** Contains all of the known router nodes. */
Hashtable nodes;
/** Contains all of the known router interfaces. */
Hashtable interfaces;
/** Contains service level pipes. */
Vector slps;
/** Describes the QoS parameters for a service level pipe. */
SLP_QoS slp_qos;
/** Contains all of the known links. */
Hashtable links;
/** Contains all of the constructed paths. */
Hashtable paths;
/** Describes the characteristics of a path. */
Path path;
/** Contains all of the assigned flows. */
Hashtable flows;
/** Describes the QoS characteristics of an assigned flow. */
Flow_QoS flow_qos;
/** Contains a sequence of service level pipes. */
Vector SLP_Sequence;
/** Describes a service level pipe. */
ServiceLevelPipe slp;
/** A boolean that will allow the showing of comments. */
private boolean showComments = false;
private SAAMRouterGui gui;

  /**
   * The <em>SLP_QoS</em> defines the QoS charateristics of a service
level pipe.
   */
  private class SLP_QoS {
    /** The maximum delay expected on this SLP. */
    int targetDelay=0;
    /** The maximum loss rate expected on this SLP. */
    int targetLossRate=0;
    /** The amount of bandwidth that this SLP should be able to provide.
*/
    int targetThroughput=0;
    /** The amount of delay being observed at this SLP. */
    int observedDelay=0;
```

```
    /** The loss rate being observed at this SLP. */
    int observedLossRate=0;
    /** The utilization being observed at this SLP. */
    int observedUtilization=0;
    public String toString(){
      return "SLP_QoS:target D="+targetDelay+", LR="+targetLossRate+",
T="
        +targetThroughput+", observed D="+observedDelay+",
LR="+observedLossRate
        +", U="+observedUtilization;
    }
  }

  /**
   * The <em>Path</em> defines the characteristics of a path.
   */
  private class Path {
    /** The first router in the path. */
    int sourceRouter=0;
    /** The last router in the path. */
    int destinationRouter=0;
    /** The total delay a flows traversing this path experiences. */
    int effectiveDelay=0;
    /** The totoal loss rate a flow traversing this path experiences. */
    int effectiveLossRate=0;
    /** The amount on bandwidth still available on this path. */
    int effectiveThroughputRemaining=0;
    /** The flows that are assigned to this path. */
    Hashtable flows = new Hashtable();
    /** The sequence of service level pipes that make up this path. */
    Vector SLPSequence = new Vector();
    public String toString(){
      return "Path: from "+sourceRouter+" to "+destinationRouter+" with
flows:"
        +flows;
    }
  }

  /**
   * The <em>Flow_QoS</em> defines the QoS characteristics of a flow.
   */
  private class Flow_QoS {
    /** The maximum amount of delay that the flow is expected to
experience. */
    int negotiatedDelay=0;
    /** The maximum loss rate that the flow is expected to experience.
*/
    int negotiatedLossRate=0;
    /** The maximum amount of bandwidth that a flow is expected to
consume. */
    int negotiatedThroughput=0;
    /** The average delay experienced by a flow. */
    int observedDelay=0;
    /** The loss rate experienced by a flow. */
    int observedLossRate=0;
    /** The amount of bandwdith being consumed by a flow. */
    int observedThroughput=0;
  }

  /**
```

```
   * The <em>ServiceLevelPipe</em> defines characteristics of a service
level
   * pipe.
   */
  private class ServiceLevelPipe {
    /** The IPv6 address of the interface. */
    IPv6Address address;
    /** The level of service that this SLP expects to provide to flows.
*/
    int serviceLevel=0;
  }


//************************************************************************
*******
// These methods are used to initialize the path information base.
//************************************************************************
******/

  /**
   * Constructs a ClassObjectStructure object that will be used to
manipulate
   * the class objects of this path information base.
   */
  public ClassObjectStructure(){
    gui = new SAAMRouterGui("PIB");
    //if (showComments){
      gui.sendText("PIB: ClassObjectStructure: Constructor executed.");
    //}
  }

  /**
   * Removes all current path data from the database.. Its most commonly
used
   * during initialization of a SAAM server for a new network.
   */
  public void deleteAllData() {
    nodes = new Hashtable();
    links = new Hashtable();
    paths = new Hashtable();
    interfaces = new Hashtable();
    if (showComments){
      gui.sendText("PIB: deleteAllData: All data deleted.");
    }
  }

//************************************************************************
*******
// These methods are used to process link state advertisements from
routers
//************************************************************************
******/

  /**
   * Determines whether a given router exists yet within the PIB.
   * @param IPv6Addresses A vector of interface addresses contained
within
   * a Hello or an LSA message.
   * @returns node_id The id of the node containing at least one of the
   * interface addresses in the vector that was passed.
   */
```

```
  public int doesRouterExist(Vector IPv6Addresses){
    Integer myNodeId;
    int node_id = 0;
    IPv6Address myIPv6Address;
    // for each of the interface IPv6 addresses that were passed in
    for (int i = 0; i < IPv6Addresses.size(); i++) {
      myIPv6Address = (IPv6Address)IPv6Addresses.elementAt(i);
      Enumeration e = nodes.keys();
      // for each of the node ids in the PIB
      while(e.hasMoreElements()){
        myNodeId = (Integer)e.nextElement();
        Hashtable myNode = (Hashtable)nodes.get(myNodeId);
        // if any of its address equals the address that was passed in
        if (myNode.containsKey(myIPv6Address.toString())){
          node_id = myNodeId.intValue();
        }
      }
    }
    if (showComments){
      if (node_id != 0)
        gui.sendText("PIB: doesRouterExist: Router " +node_id+ "
exists.");
      else
        gui.sendText("PIB: doesRouterExist: Router is not in
database.");
    }
    return node_id;
  }

  /**
   * Finds an unassigned node id and adds it to the PIB. It is commonly
used
   * for assigning a new node_id to a previously unknown router.
   * @returns max_node_id An unassigned router id.
   */
  public int getNewNodeId(){
    int max_node_id = 0;
    Enumeration e = nodes.keys();
    // for each of the node ids in the PIB
    while(e.hasMoreElements()){
      Integer myNodeId = (Integer)e.nextElement();
      // if the id is greater than the max
      if (max_node_id < myNodeId.intValue())
        // then assign it as the max
        max_node_id = myNodeId.intValue();
    }
    // increment the max to get a new max
    max_node_id++;
    // enter this node id into the PIB
    nodes.put(new Integer(max_node_id), new Hashtable());
    if (showComments){
      gui.sendText("PIB: assignNewNodeId: Router's id assigned: "
        + max_node_id);
    }
    return max_node_id;
  }

  /**
   * Determines whether a given interface exists yet within the PIB.
   * @param myIPv6Address The interface address contained within a hello
or LSA
```

```
     * message.
     * @returns found True if the interface address already exists within
the PIB.
     */
    public boolean doesInterfaceExist(IPv6Address myIPv6Address){
      boolean found = false;
      Integer myNodeId;
      Enumeration e = nodes.keys();
      // for each node in the PIB
      while(e.hasMoreElements()){
        myNodeId = (Integer)e.nextElement();
        Hashtable myNode = (Hashtable)nodes.get(myNodeId);
        // if any of its interface addresses equal LSA interface address
        if (myNode.containsKey(myIPv6Address.toString()))
          found = true;
      }
      if (showComments){
        if (found)
          gui.sendText("PIB: doesInterfaceExist: Interface "
            + myIPv6Address.toString() + " is found.");
        else
          gui.sendText("PIB: doesInterfaceExist: Interface "
            + myIPv6Address.toString() + " is not found.");
      }
      return found;
    }

    /**
     * Determines whether a given link exists yet within the PIB.
     * @param address The IPv6 address of an interface.
     * @returns found True if the link address already exists within the
PIB.
     */
    public boolean doesLinkExist(IPv6Address address){
      boolean found = false;
      // if the links known to the PIB contains this address
      if (links.containsKey(address.getNetworkAddress().toString()))
        found = true;
      if (showComments){
        if (found)
          gui.sendText("PIB: doesLinkExist: Link "
            + address.getNetworkAddress() + " is found.");
        else
          gui.sendText("PIB: doesLinkExist: Link "
            + address.getNetworkAddress() + " is not found.");
      }
      return found;
    }

    /**
     * Adds a new link to the PIB.
     * @param address The IPv6 address of an interface.
     * @param max_bandwidth The max transmission rate over this network
segment.
     */
    public void addLink(IPv6Address address, int max_bandwidth){
      // add the link entry to the hash table
      links.put(address.getNetworkAddress().toString(),new
Integer(max_bandwidth));
      if (showComments){
        gui.sendText("PIB: addLink: Link " + address.getNetworkAddress()
```

```
        + " is added.");
    }
  }

  /**
   * Adds a new interface to the PIB.
   * @param node_id The id of the router whose interface is being added.
   * @param address The IPv6 address of an interface.
   */
  public void addInterface(int node_id, IPv6Address address){
    // get the node assigned to this node id
    Hashtable myNode = (Hashtable)nodes.get(new Integer(node_id));
    // add this new interface to the node
    myNode.put(address.toString(), new Vector());
    if (showComments){
      gui.sendText("PIB: addInterface: Interface "+ address +" is
added.");
    }
  }

  /**
   * Determines whether a service level pipe exists yet within the PIB.
   * @param address The IPv6 address of an interface.
   * @param service_level The level of service that this logical pipe is
   * providing.
   * @returns found True if this SLP is already in the PIB.
   */
  public boolean doesSLPExist(IPv6Address myIPv6Address, int
service_level){
    boolean found = false;
    Integer myNodeId = new Integer(0);
    Enumeration e_node_ids = nodes.keys();
    // for each of the node ids in the PIB
    while(e_node_ids.hasMoreElements()){
      myNodeId = (Integer)e_node_ids.nextElement();
      Hashtable Interfaces = (Hashtable)nodes.get(myNodeId);
      // if this interface's address equals the interface address of
this slp
      if (Interfaces.containsKey(myIPv6Address.toString())){
        Vector slps = (Vector)Interfaces.get(myIPv6Address.toString());
        // if this interface has more service levels than this service
level
        if (slps.size() >= service_level) {
          found = true;
        }
      }
    }
    if (showComments){
      if (found)
        gui.sendText("PIB: doesSLPExist: SLP from router "
          + myNodeId + " is found.");
      else
        gui.sendText("PIB: doesSLPExist: SLP from router "
          + myNodeId + " is not found.");
    }
    return found;
  }

  /**
   * Updates the status of a known SLP's delay, loss_rate, and
throughput.
```

```
   * @param address The IPv6 address of an interface.
   * @param service_level The level of service that this logical pipe is
   * providing.
   * @param delay The average delay experienced by a packet's stay in
the
   * particular SLP outbound queue.
   * @param loss_rate The average loss_rate experienced by packets in a
   * particular SLP.
   * @param throughput The average throughput provided by a particular
SLP.
   */
  public void updateSLP(IPv6Address address, int service_level, int
delay,
     int loss_rate, int throughput){
     Integer myNodeId = new Integer(0);
     Enumeration e = nodes.keys();
     // for each of the nodes in the PIB
     while(e.hasMoreElements()){
       myNodeId = (Integer)e.nextElement();
       Hashtable myNode = (Hashtable)nodes.get(myNodeId);
       // if this node contains an interface with the address passed in
       if (myNode.containsKey(address.toString())){
         Vector myInterface = (Vector)myNode.get(address.toString());
         // if this interface has more service levels than this service
level
         if (myInterface.size() >= service_level) {
           slp_qos = (SLP_QoS)myInterface.elementAt(service_level);
           // update it with these new values
           slp_qos.observedDelay = delay;
           slp_qos.observedDelay = loss_rate;
           slp_qos.observedDelay = throughput;
           myInterface.insertElementAt(slp_qos,service_level);
           if (showComments){
             gui.sendText("PIB: updateSLP: SLP " + service_level
               + " is assigned delay="+delay+",loss_rate="+loss_rate
               + ",throughput="+throughput);
           }
         }
       }
     }
     if (showComments){
       gui.sendText("PIB: updateSLP: SLP " + service_level +" is
updated.");
     }
  }

  /**
   * Adds a previously unknown SLP to the PIB along with its targeted
QoS.
   * @param address The IPv6 address of an interface.
   * @param service_level The level of service that this logical pipe is
   * providing.
   * @param target_delay The average delay experienced by a packet's
stay in the
   * particular SLP outbound queue.
   * @param target_loss_rate The average loss_rate experienced by
packets in a
   * particular SLP.
   * @param target_throughput The average throughput provided by a
particular SLP.
   */
```

```
   public void addSLP(IPv6Address address, int service_level, int
 target_delay,
     int target_loss_rate, int target_throughput){
     int delay = 0, loss_rate = 0, throughput = 0;
     Integer myNodeId = new Integer(0);
     Enumeration e = nodes.keys();
     // for each node in the PIB
     while(e.hasMoreElements()){
       myNodeId = (Integer)e.nextElement();
       Hashtable myNode = (Hashtable)nodes.get(myNodeId);
       // if any interface has this same IPv6 address
       if (myNode.containsKey(address.toString())){
         Vector myInterface = (Vector)myNode.get(address.toString());
         SLP_QoS slp_qos = new SLP_QoS();
         slp_qos.targetDelay = target_delay;
         slp_qos.targetLossRate = target_loss_rate;
         slp_qos.targetThroughput = target_throughput;
         slp_qos.observedDelay = delay;
         slp_qos.observedLossRate = loss_rate;
         slp_qos.observedUtilization = throughput;
         // add this new slp to this interface
         myInterface.insertElementAt(slp_qos,service_level);
       }
     }
     if (showComments){
       gui.sendText("PIB: addSLP: SLP " + service_level + " is added to "
         +address);
     }
   }

//**********************************************************************
*******
// These methods are used to process a flow request from a host
//**********************************************************************
******/

  /**
   * Finds a router id that has an interface to on the same link as the
host
   * making a flow request.
   * @param address The IPv6 address of the interface of the host
requesting
   * the flow.
   * @returns ARouter The router id of the first router found on this
link.
   */
  public int findARouterOnLink(IPv6Address address){
    int ARouter = 0;
    Integer myNodeId = null;
    // check to see if requesting host is a router itself
   Enumeration e = nodes.keys();
    // for each of the node ids in the PIB
    while(e.hasMoreElements()){
      myNodeId = (Integer)e.nextElement();
      Hashtable myNode = (Hashtable)nodes.get(myNodeId);
      // if any of its address equals the address that was passed in
      if(myNode.containsKey(address.toString())){
        ARouter = myNodeId.intValue();
      }
    }
    // otherwise, find any other router on the same subnet
```

```
    if (ARouter == 0){
      e = nodes.keys();
      // for each of the node ids in the PIB
      while(e.hasMoreElements()){
        myNodeId = (Integer)e.nextElement();
        Hashtable myNode = (Hashtable)nodes.get(myNodeId);
        Enumeration addresses = myNode.keys();
        // for each of these interfaces
        while (addresses.hasMoreElements()){
          String nextAddress = (String)addresses.nextElement();
          try{
            // if this interface's network address equals that of the
link
            if
(IPv6Address.getByName(nextAddress).getNetworkAddress().toString().equal
s(

address.getNetworkAddress().toString())){
              ARouter = myNodeId.intValue();
            }
          }catch(UnknownHostException uhe){
            gui.sendText(""+uhe);
          }
        }
      }
    }
    if (showComments){
      gui.sendText("PIB: findARouterOnLink: Router "
        + ARouter + " is found on same link " +
address.getNetworkAddress()
        + " as host " + address);
    }
    return ARouter;
  }

  /**
   * Determines if there is a path that can support a particular flow
request.
   * A value of zero is returned if no path can support this QoS.
   * @param source_router The node id of a router on the same physical
link as
   * the source host.
   * @param destination_router The node id of a router on the same
physical
   * link as the destination host.
   * @param myFlowRequest A host's request for the establishment of a
flow.
   * @returns path_id The id of a path that can support this request.
   */
  public int getPathThatCanSupportFlowRequest(int source_router,
                       int destination_router, FlowRequest
myFlowRequest){
    int path_id = 0;
    // for each path in the PIB
    Enumeration e_path_ids = paths.keys();
    while (e_path_ids.hasMoreElements()){
      Integer nextPathId = (Integer)e_path_ids.nextElement();
      Path nextPath = (Path)paths.get(nextPathId);
      // if it has the same source and destination router
      //   and its effective delay and loss rate is less than this
request
```

```
      //   and its throughput remaining is more than the requested
throughput
      if (nextPath.sourceRouter == source_router &&
          nextPath.destinationRouter == destination_router &&
          nextPath.effectiveDelay <= myFlowRequest.getRequestedDelay()
&&
          nextPath.effectiveLossRate <=
myFlowRequest.getRequestedLossRate() &&
          nextPath.effectiveThroughputRemaining
                                        >=
myFlowRequest.getRequestedThroughput()){
        path_id = nextPathId.intValue();
      }
    }
    if (showComments){
      if (path_id != 0){
        gui.sendText("PIB: getPathThatCanSupportFlowRequest: "
          + "Flow request from "+source_router+" to "+destination_router
          +" with delay<="+myFlowRequest.getRequestedDelay()+", LR<="
          +myFlowRequest.getRequestedLossRate()+", RT>="
          +myFlowRequest.getRequestedThroughput()+"can be supported on
path: "
          + path_id);
      }
      else{
        gui.sendText("PIB: getPathThatCanSupportFlowRequest: "
          + "Flow request cannot be supported.");
      }
    }
    return path_id;
  }

  /**
   * Finds an unassigned flow id and assigns this new flow to a path.
   * @param path_id The id of a path that can support this request.
   * @param source_router The node id of a router on the same physical
link as
   * the source host.
   * @param destination_router The node id of a router on the same
physical
   * link as the destination host.
   * @param myFlowRequest A host's request for the establishment of a
flow.
   * @returns max_flow_id The id that is being assigned to this flow.
   */
  public int getNewFlowId(int path_id, int source_router,
                             int destination_router, FlowRequest
myFlowRequest){
    int max_flow_id = 0;
    Enumeration e_path_ids = paths.keys();
    // for each path in the PIB
    while (e_path_ids.hasMoreElements()){
      Integer nextPathId = (Integer)e_path_ids.nextElement();
      Path nextPath = (Path)paths.get(nextPathId);
     Enumeration e_flow_ids = nextPath.flows.keys();
      // for each of the flows on this path
      while (e_flow_ids.hasMoreElements()){
        Integer nextFlowId = (Integer)e_flow_ids.nextElement();
        // if this flow id is greater than the current max flow id
        if (nextFlowId.intValue() > max_flow_id){
          max_flow_id = nextFlowId.intValue();
```

```
          }
        }
      }
      // now increment to get an unassigned flow id
      max_flow_id++;

      // now assign this flow to this path
      Path myPath = (Path)paths.get(new Integer(path_id));
      Flow_QoS myFlowQoS = new Flow_QoS();
      myFlowQoS.negotiatedDelay = myFlowRequest.getRequestedDelay();
      myFlowQoS.negotiatedDelay = myFlowRequest.getRequestedLossRate();
      myFlowQoS.negotiatedDelay = myFlowRequest.getRequestedThroughput();
      myPath.flows.put(new Integer(max_flow_id),myFlowQoS);
      if (showComments){
        gui.sendText("PIB: assignNewFlowId: Flow "
          + max_flow_id + " is assigned to path "+path_id);
      }
      return max_flow_id;
    }

    /**
     * Retrieves the sequence of SLP that make up a given path.
     * @param path_id The id of the path in question.
     * @returns slps_in_path A vector of SLPs that compose this path.
     */
    public Vector getSLPSequenceOfPath(int path_id){
      int sequenceNumber = 0;
      IPv6Address link_id = null;
      Vector slps_in_path = new Vector();
      // get the sequence of slps that compose the path
      Path path = (Path)paths.get(new Integer(path_id));
      // for each slp
      for (int index = 0; index < path.SLPSequence.size(); index++){
        ServiceLevelPipe nextSLP =
(ServiceLevelPipe)path.SLPSequence.elementAt(index);
        // instantiate a slp sequence object
        SLPSequence SLP_Sequence = new SLPSequence();
        // set the slp sequence values
        SLP_Sequence.setServiceLevel(nextSLP.serviceLevel);
        // find what node this slp is attached to...
        Enumeration e = nodes.keys();
        int node_id = 0;
        // for each of the node ids in the PIB
        while(e.hasMoreElements()){
          Integer myNodeId = (Integer)e.nextElement();
          Hashtable myNode = (Hashtable)nodes.get(myNodeId);
          if (myNode.containsKey(nextSLP.address.toString())){
            node_id = myNodeId.intValue();
            link_id = nextSLP.address.getNetworkAddress();
          }
        }

        SLP_Sequence.setSourceRouter(node_id);
        SLP_Sequence.setPathId(path_id);
        SLP_Sequence.setLinkId(link_id);
        SLP_Sequence.setSequenceNumber(sequenceNumber);
        if (showComments){
          gui.sendText("PIB: getSLPSequenceOfPath: adding "+SLP_Sequence);
        }
        // add it to the vector
        slps_in_path.addElement(SLP_Sequence);
```

```
        // increment for the next slp
        sequenceNumber++;
      }
      if (showComments){
        gui.sendText("PIB: getSLPSequenceOfPath: SLP sequence of path
"+path_id
          +" has "+path.SLPSequence.size()+" hops.");
      }
      return slps_in_path;
    }

    /**
     * Retrieves  the IPv6 address of an interface.
     * @param node_id The id of the router whose interface is being
queried.
     * @param link_id The network portion of the IPv6 address of an
interface.
     * @returns address The address of the interface that connects this
node and
     * link.
     */
    public IPv6Address getInterfaceAddress(int node_id, IPv6Address
link_id){
      IPv6Address address = new IPv6Address();
      IPv6Address tempAddress = null;
      String nextAddress;
      Hashtable node_interfaces = (Hashtable)nodes.get(new
Integer(node_id));
      Enumeration e_addresses = node_interfaces.keys();
      // for each of these interfaces
      while (e_addresses.hasMoreElements()){
        nextAddress = (String)e_addresses.nextElement();
        try{
          tempAddress = IPv6Address.getByName(nextAddress);
        }catch(UnknownHostException uhe){
          gui.sendText(""+uhe);
        }
        // if this interface's network address equals that of the link
        if
(tempAddress.getNetworkAddress().toString().equals(link_id.toString())){
          address = tempAddress;
        }
      }
      if (showComments){
        gui.sendText("PIB: getInterfaceAddress: Interface " + address
          + " connects " + node_id + " to link " + link_id);
      }
      return address;
    }

//***********************************************************************
*******
// These methods are used to determine all possible paths across the
network
//***********************************************************************
******/

    /**
     * Retrieve all of the router ids assigned by the PIB so far.
     * @returns V A vector of all assigned router ids.
     */
```

```
public Vector getAllRouterIds(){
  Vector V = new Vector();
  Integer myNodeId;
  Enumeration e = nodes.keys();
  // for each node id in the PIB
  while(e.hasMoreElements()){
    myNodeId = (Integer)e.nextElement();
    // add it to the vector
    V.addElement(myNodeId);
  }
  if (showComments){
    gui.sendText("PIB: getAllRouterIds: All router ids returned:");
    gui.sendText(""+V);
  }
  return V;
}


/**
 * Retrieves the maximum service level of this SAAM region.
 * @returns max_slp_id The numerically highest service level id
assigned.
 */
public int findMaxServiceLevel(){
  int max_slp_id = 0;
  Hashtable myNode = new Hashtable();
  Enumeration e1,e2;
  e1 = nodes.elements();
  // for each node in the PIB
  while(e1.hasMoreElements()){
    myNode = (Hashtable)e1.nextElement();
    e2 = myNode.elements();
    // for each interface of this node
    if (e2.hasMoreElements()){
      Vector myInterface = (Vector)e2.nextElement();
      // determine the maximum number of service levels
      if (max_slp_id < myInterface.size())
        max_slp_id = myInterface.size()-1;
    }
  }
  if (showComments){
    gui.sendText("PIB: findMaxServiceLevel: The max service level is "
      + max_slp_id);
  }
  return max_slp_id;
}

/**
 * Retrieves an array of parents for each router.  A parent is a
directly
 * connected node.
 * @param V A vector of all router ids.
 * @param service_level The level of service for which paths are being
built
 * for.
 * @returns parent A hashtable of vectors containing the parents of
each router.
 */
public Hashtable getParents(Vector V, int service_level){
  Hashtable node1 = new Hashtable();
  Hashtable node2 = new Hashtable();
```

```
    Vector myVector = new Vector();
    Enumeration e1_interface_ids,e2_nodes, e2_node_ids,e3_interface_ids;
    IPv6Address link_id = null;
    Integer node_id;
    String address;
    Hashtable parent = new Hashtable();
    // for each "destination" node in vector V
    for (int index = 0; index < V.size(); index++){
      myVector = new Vector();
      // get this destination node's interfaces in order to know
      //   all of its directly connected links
      node1 = (Hashtable)nodes.get(V.elementAt(index));
      e1_interface_ids = node1.keys();
      // for each interface ( or directly connected link...)
      while (e1_interface_ids.hasMoreElements()){
        try{
          link_id = IPv6Address.getByName(
((String)e1_interface_ids.nextElement())).getNetworkAddress();
        }catch(UnknownHostException uhe){
          gui.sendText(""+uhe);
        }
        //get all nodes that that are also connected to the link
        e2_nodes = nodes.elements();
        e2_node_ids = nodes.keys();
        // for each node in the PIB
        while (e2_nodes.hasMoreElements()){
          node2 = (Hashtable)e2_nodes.nextElement();
          node_id = (Integer)e2_node_ids.nextElement();
          e3_interface_ids = node2.keys();
         // for each interface on this node
          while(e3_interface_ids.hasMoreElements()){
            address = (String)e3_interface_ids.nextElement();
           try{
              IPv6Address tempAddress =
IPv6Address.getByName(address).getNetworkAddress();
              // if this interface is also connected to this link
              //   and this node is not the "destination" node
              if ((link_id.toString().equals(tempAddress.toString()))
                                                  &&
!node_id.equals(V.elementAt(index))){
                // add it to the parent vector of this "destination"
node
                myVector.addElement(node_id);
              } //end if
            }catch(UnknownHostException uhe){
              gui.sendText(""+uhe);
            }
          } // end while e3_interface_ids
        } // end while e2_nodes
      } // end while e1_interfaces
      parent.put(V.elementAt(index),myVector);
    } // end for
    if (showComments){
      gui.sendText("PIB: getParents: Parent hashtable returned:");
      gui.sendText(""+parent);
    }
    return parent;
  }

  /**
```

```
   * Assigns a path from a source and to a destination.
   * @param source_router The node_id of the source of the path.
   * @param destination_router The node_id of the destination of the
path.
   * @returns max_path_id The id to be assigned to this new path.
   */
  public int getNewPathId(int source_router,int destination_router){
    int max_path_id = 0;
    Integer path_id;
    Enumeration e_path_ids = paths.keys();
    // for each path in the PIB
    while (e_path_ids.hasMoreElements()){
      path_id = (Integer)e_path_ids.nextElement();
      // if this path id is greater than the max so far
      if (max_path_id < path_id.intValue())
        max_path_id = path_id.intValue();
    }
    // increment to get unassigned path id
    max_path_id++;
    Path path = new Path();
    path.sourceRouter = source_router;
    path.destinationRouter = destination_router;
    // add this new path to the PIB
    paths.put(new Integer(max_path_id),path);
    if (showComments){
      gui.sendText("PIB: getNewPathId: Path " + max_path_id + " from "
        +source_router+" to "+destination_router+" is inserted.");
    }
    return max_path_id;
  }

  /**
   * Identifies the id of the physical link between two adjacent
routers. If no
   * link exists between a source and destination router, the default
address
   * of all zeros is returned.
   * @param source_router The node_id of a router.
   * @param destination_router The node_id of an adjacent router.
   * @returns subnet The link id between this source and destination.
   */
  public IPv6Address getLinkBetween(int source_router, int
destination_router){
    IPv6Address subnet = new IPv6Address();
    String source_address, destination_address;
    Hashtable source_node = (Hashtable)nodes.get(new
Integer(source_router));
    Hashtable destination_node =
                         (Hashtable)nodes.get(new
Integer(destination_router));
    Enumeration e_source_interfaces = source_node.keys();
    // for each interface assigned to this source node in the PIB
    while (e_source_interfaces.hasMoreElements()){
      source_address = (String)e_source_interfaces.nextElement();
      Enumeration e_destination_interfaces = destination_node.keys();
      // for each interface assigned to this destination node in the PIB
      while (e_destination_interfaces.hasMoreElements()){
        destination_address =

(String)e_destination_interfaces.nextElement();
        try{
```

```
          // if this destination interface equals this source interface
          if
(IPv6Address.getByName(source_address).getNetworkAddress().toString().eq
uals(

IPv6Address.getByName(destination_address).getNetworkAddress().toString(
))){
            // get the network address
            subnet =
IPv6Address.getByName(source_address).getNetworkAddress();
          }
        }catch (UnknownHostException uhe){
          gui.sendText(""+uhe);
        }
      }
    }
    if (showComments){
      gui.sendText("PIB: getLinkBetween: Link between " + source_router
        + " and " + destination_router + " is " + subnet);
    }
    return subnet;
  }

  /**
   * Assigns a service level pipe sequence entry in the building of a
path.
   * @param service_level The level of service for which paths are being
built
   * for.
   * @param source_router The node_id of the source of the SLP.
   * @param link_id The subnet that this SLP goes over.
   * @param path_id The id assigned to the path.
   * @param sequence_number The number assigned to specify the sequence
of this
   * SLP in the path.
   */
  public void assignSLPSequence(int service_level, int source_router,
    IPv6Address link_id, int path_id, int sequence_number){
    // get this path object
    Path myPath = (Path)paths.get(new Integer(path_id));
    ServiceLevelPipe slp = new ServiceLevelPipe();
    slp.address = getInterfaceAddress(source_router, link_id);
    slp.serviceLevel = service_level;
    // add this slp to the sequence of slps in this path
    myPath.SLPSequence.insertElementAt(slp, sequence_number);
    if (showComments){
      gui.sendText("PIB: assignSLPSequence: SLP#"+service_level+" from "
        + source_router + " to link " + link_id + " on path " + path_id
        + " is assigned sequence number " + sequence_number);
    }
  }

//**********************************************************************
*******
// These methods are used to determine the effective QoS of paths
//**********************************************************************
******/

  /**
   * Retrieves a vector of all path ids constructed by the PIB.
   * @returns path_ids All of the path ids known to the PIB.
```

```
  */
public Vector getAllPathIds(){
  Vector path_ids = new Vector();
  Enumeration e_path_ids = paths.keys();
  // for each path
  while (e_path_ids.hasMoreElements()){
    // add it to vector
    path_ids.addElement(e_path_ids.nextElement());
  }
  if (showComments){
    gui.sendText("PIB: getAllPathIds: paths in PIB:");
    gui.sendText(""+path_ids);
  }
  return path_ids;
}

/**
 * Retrieves the SLPs that make up a given path.
 * @param path_id The id of the path in question.
 * @returns slps_in_path The SLPs that make up the path.
 */
public Vector getSLPsOfPath(int path_id){
  Vector slps_in_path = new Vector();
  SLP mySLP = new SLP();
  int node_id = 0;
  Path path = (Path)paths.get(new Integer(path_id));
  // for each of the slps in this path
  for (int index = 0; index < path.SLPSequence.size(); index++){
    ServiceLevelPipe slp =
(ServiceLevelPipe)path.SLPSequence.elementAt(index);
    // set the values for delivery to the server
    mySLP.setServiceLevel(slp.serviceLevel);
    mySLP.setAddress(slp.address);
    Enumeration e = nodes.keys();
    // for each of the nodes in the PIB
    while(e.hasMoreElements()){
      Integer myNodeId = (Integer)e.nextElement();
      Hashtable myNode = (Hashtable)nodes.get(myNodeId);
      // if this node contains this interface IPv6 address
      if (myNode.containsKey(slp.address.toString()))
        node_id = myNodeId.intValue();
    }
    // with this node's id, get its interfaces
    Hashtable interfaces = (Hashtable)nodes.get(new Integer(node_id));
    // get vector of slp_qos
    Vector slps = (Vector)interfaces.get(slp.address.toString());
    // get slp_qos for this particular slp
    SLP_QoS slp_qos = (SLP_QoS)slps.elementAt(slp.serviceLevel);
    // set QoS of this slp for delivery to the server
    mySLP.setTargetThroughput(slp_qos.targetThroughput);
    mySLP.setDelay(slp_qos.observedDelay);
    mySLP.setLossRate(slp_qos.observedLossRate);
    int observedThroughput = slp_qos.observedUtilization / 100
                                                     *
slp_qos.targetThroughput /10;
    mySLP.setThroughput(observedThroughput);
    if (showComments){
      gui.sendText("PIB:getSLPsOfPath: path " + path_id + ": address "
        +slp.address+", service_level "+slp.serviceLevel+" observed
values:");
      gui.sendText("PIB:getSLPsOfPath: D = "+slp_qos.observedDelay
```

```
          +"ms, LR = " +slp_qos.observedLossRate+"%, U = "
          + slp_qos.observedUtilization / 10
          + "%, T = "+observedThroughput+"kbps");
      }
      // add this slp to the vector for delivery to the server
      slps_in_path.addElement(mySLP);
    }
    if (showComments){
      gui.sendText("PIB: getSLPsOfPath: SLP in path " + path_id +":");
      gui.sendText(""+slps_in_path);
    }
    return slps_in_path;
  }


  /**
   * Records the calculated effective quality of service parameters for
a
   * particular path.
   * @param path_id The id of the path in question.
   * @param effectiveDelay The effective delay that can be expected when
   * transmiting a flow over this path.
   * @param effectiveLossRate The effective loss rate that can be
expected when
   * transmiting a flow over this path.
   * @param effectiveThroughputRemaining The effective throughput
capacity that
   * was not being used at last observation.
   */
  public void setEffectiveQoSOfPath(int path_id, int effectiveDelay,
    int effectiveLossRate, int effectiveThroughputRemaining){
    // get this path
    Path path = (Path)paths.get(new Integer(path_id));
    // set its effective QoS
    path.effectiveDelay = effectiveDelay;
    path.effectiveLossRate = effectiveLossRate;
    path.effectiveThroughputRemaining = effectiveThroughputRemaining;
    if (showComments){
      gui.sendText("PIB: setEffectiveQoSOfPath: path "+path_id
        +" effectively has D = " + effectiveDelay
        + "ms, LR = " + effectiveLossRate
        + "%, remaining T = " + effectiveThroughputRemaining+"kbps");
    }
  }

  /**
   * Retrieves a vector of all path ids that travses the specified SLP.
   * @param address The IPv6 address of an interface.
   * @param service_level The level of service that this logical pipe is
   * providing.
   * @returns path_ids All of the path ids that traverse this SLP.
   */
  public Vector getAllPathIdsThatTraverseSLP(IPv6Address address,
                                                              int
service_level){
    Vector path_ids = new Vector();
    Enumeration e_paths = paths.elements();
    Enumeration e_path_ids = paths.keys();
    // for each path
    while (e_paths.hasMoreElements()){
      Path path = (Path)e_path_ids.nextElement();
```

```
      Integer path_id = (Integer)e_path_ids.nextElement();
      // get the sequence of slps that make up this path
      Vector slps = path.SLPSequence;
      // for each of each of these slps
      for (int index = 0; index < slps.size(); index++){
        ServiceLevelPipe slp = (ServiceLevelPipe)slps.elementAt(index);
        // if this slp's interface address equals this address
        if (slp.address.equals(address)){
          // add it to the vector
          path_ids.addElement(path_id);
        }
      }
    }
    if (showComments){
      gui.sendText("PIB: getAllPathIdsThatTravseSLP: paths over
"+address
        +"'s service level #"+ service_level);
      gui.sendText(""+path_ids);
    }
    return path_ids;
  }

//*********************************************************************
*******
// These methods are used to retrieve various other info
//*********************************************************************
******/

  /**
   * Retrieves a vector of all interface addresses attached to this
router.
   * @param node_id The node id of the router in question.
   * @returns IPv6Addresses The interface addresses of this node.
   */
  public Vector getRouterInterfaces(int node_id){
    Vector IPv6Addresses = new Vector();
    Hashtable myNode = (Hashtable)nodes.get(new Integer(node_id));
    Enumeration e = myNode.keys();
    // for each interface on this node
    while(e.hasMoreElements()){
      String myAddress = (String)e.nextElement();
      try{
        // add it to the vector
        IPv6Addresses.addElement(IPv6Address.getByName(myAddress));
      }catch(UnknownHostException uhe){
        gui.sendText(""+uhe);
      }
    }
    if (showComments){
      gui.sendText("PIB: getRouterInterfaces: interfaces of node "
        +node_id+" returned:");
      gui.sendText(""+IPv6Addresses);
    }
    return IPv6Addresses;
  }

  /**
   * Deletes a specified router from the PIB.
   * @param node_id The node_id of the router to be deleted.
   */
  public void deleteARouter(int node_id){
```

```
      nodes.remove(new Integer(node_id));
      if (showComments){
        gui.sendText("PIB: deleteARouter: node "+node_id+" deleted.");
      }
    }

    /**
     * Retrieves a vector of all physical link ids known to the PIB.
     * @returns v_links All of the known links in the network.
     */
    public Vector getAllLinkIds(){
      Vector v_links = new Vector();
      Enumeration e = links.keys();
      // for each interface on this node
      while(e.hasMoreElements()){
        String myAddress = (String)e.nextElement();
        try{
          // add it to the vector
          v_links.addElement(IPv6Address.getByName(myAddress));
        }catch(UnknownHostException uhe){
          gui.sendText(""+uhe);
        }
      }
      if (showComments){
        gui.sendText("PIB: getAllLinkIds: all link ids:");
        gui.sendText(""+v_links);
      }
      return v_links;
    }

    /**
     * Retrieves a vector of all routers attached to a specific phyical
  link.
     * @param link_id The IPv6 address of the link in question.
     * @returns routerIds The ids of routers that are directly attached to
  this
     * link.
     */
    public Vector findRoutersOnLink(IPv6Address link_id){
      Vector routerIds = new Vector();
      Enumeration e = nodes.keys();
      // for each of the node ids in the PIB
      while(e.hasMoreElements()){
        Integer myNodeId = (Integer)e.nextElement();
        Hashtable myNode = (Hashtable)nodes.get(myNodeId);
        Enumeration addresses = myNode.keys();
        // for each of these interfaces
        while (addresses.hasMoreElements()){
          String nextAddress = (String)addresses.nextElement();
          try{
            // if this interface's network address equals that of the link
            if
  (IPv6Address.getByName(nextAddress).getNetworkAddress().toString()

  .equals(link_id.toString())){
              routerIds.addElement(myNodeId);
            }
          }catch(UnknownHostException uhe){
            gui.sendText(""+uhe);
          }
        }
```

```
    }
    if (showComments){
      gui.sendText("PIB: findRoutersOnLink:routers on
link"+link_id+":");
      gui.sendText(""+routerIds);
    }
    return routerIds;
  }

}
```

```
package saam.server;

import saam.*;
import saam.event.*;
import saam.net.*;
import saam.message.*;
import java.net.*;
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * The <em>Console</em> is an object within the SAAM architecture that
 * provides a display of the inner status of the SAMM server and of the network
 * in general.
 */

public class Console extends HttpServlet {

  /** The object which the Console displays the status of. */
  Server myServer = new Server("database");

  /** The number of interfaces that the user defines a router as having. */
  int NumOfInterfacesToAdd = 0;

  /**
   * The servlet engine calls the <em>init</em> method exactly once on a
   * servlet, after the servlet is instantiated but before it is placed into
   * service. The init method must exit successfully before you can call the
   * service method.
   * @ param config Supplies servlet with initialization parameters.
   */
  public void init(ServletConfig config) throws ServletException {
    super.init(config);
    try{
        Translator t = new Translator(9001,65508,9002,65508);
    }catch(SocketException se){
      System.out.println(se.toString());
    }//try-catch
  }

  /**
   * The web server invokes this method each time it receives a GET request for
   * this servlet.
   * @param req Represents the client's request.
   * @param res Represents the servlet's response.
   */
  public void doGet(HttpServletRequest req, HttpServletResponse res)
                              throws ServletException, IOException {

    if (req.getMethod().equals("HEAD")) return;

    String action = req.getParameter("action");
    String numberOfInterfaces;
    int intNumOfInterfaces = 0;
    try{
    if (action.equals("defineRouter")) {
      defineRouter(res);
    }
    else {
```

```
    if (action.equals("defineNumOfInterfaces")) {
      numberOfInterfaces = req.getParameter("numOfInterfaces");
      NumOfInterfacesToAdd = (new Integer(numberOfInterfaces)).intValue();
      defineNumOfInterfaces(NumOfInterfacesToAdd, res);
    }
    else {
    if (action.equals("defineInterfaces")) {
      defineInterfaces(NumOfInterfacesToAdd, req, res);
    }
    else {
    if (action.equals("displayRouters")) {
      displayRouters(res);
    }
    else {
    if (action.equals("selectRouterForDeletion")) {
      selectRouterForDeletion(res);
    }
    else {
    if (action.equals("deleteARouter")) {
      deleteARouter(req,res);
    }
    else {
    if (action.equals("selectALink")) {
      selectALink(res);
    }
    else {
    if (action.equals("findRoutersOnLink")) {
      findRoutersOnLink(req, res);
    }
    else {
      displayConsole(res);
    }}}}}}}}
    }catch (NullPointerException npe){
      displayConsole(res);
    }
  }

  /**
   * Displays the main console page from which a user selects what activity to
   * perform.
   * @param res Represents the servlet's response.
   */
  public void displayConsole(HttpServletResponse res){
    res.setContentType("text/html");
    try{
      PrintWriter out = res.getWriter();
      out.println("<HTML><HEAD><TITLE>SAAM Server Console</TITLE>"
        + "<LINK REL=STYLESHEET TYPE=\"text/css\" HREF=\"/core-styles.css\">"
        + "</HEAD>");
      out.println("<BODY><H1>Welcome to the SAAM Server Console</H1>");
      out.println("<H2>What would you like to do? </H2>");
      out.println("<FORM METHOD=GET ACTION=\"/servlet/saam.server.Console\">"
        + "<P class=\"component\">"
        + "<INPUT TYPE=RADIO NAME=\"action\" VALUE=\"defineRouter\">"
        + "Define a router</P>"
        + "<P class=\"component\">"
        + "<INPUT TYPE=RADIO NAME=\"action\" VALUE=\"displayRouters\">"
        + "Display all routers</P>"
        + "<P class=\"component\">"
        + "<INPUT TYPE=RADIO NAME=\"action\" VALUE=\"selectALink\">"
        + "Find all routers on a subnet</P>"
```

```
        + "<P class=\"component\">"
        +"<INPUT TYPE=RADIO NAME=\"action\" VALUE=\"selectRouterForDeletion\">"
        + "Delete a router</P>"
        + "<P class=\"component\"><INPUT TYPE=SUBMIT VALUE=Enter><P>");
      out.println("</FORM></BODY><HTML>");
    }catch(IOException ioe) {
      System.err.println(ioe);
    }
  }

  /**
   * Queries a user to begin defining a router by specifying how many
   * interfaces are present on the router.
   * @param res Represents the servlet's response.
   */
  public void defineRouter(HttpServletResponse res){
    res.setContentType("text/html");
    try{
      PrintWriter out = res.getWriter();
      out.println("<HTML><HEAD><TITLE>Define A Router</TITLE>"
        + "<LINK REL=STYLESHEET TYPE=\"text/css\" HREF=\"/core-styles.css\">"
        + "</HEAD>");
      out.println("<BODY><H1>Define a router: </H1>");
      out.println("<FORM METHOD=GET ACTION=\"/servlet/saam.server.Console\">"
        + "<INPUT TYPE=HIDDEN NAME=\"action\" VALUE=\"defineNumOfInterfaces\">"
        + "<P>"
        + "<P class=\"component\">How many interfaces are on this router?"
        + "<INPUT TYPE=TEXT NAME=\"numOfInterfaces\"></P>"
        + "<INPUT TYPE=SUBMIT VALUE=Enter><P>");
      out.println("</FORM></BODY><HTML>");
    }catch(IOException ioe) {
      System.err.println(ioe);
    }
  }

  /**
   * After the user defines the number of interfaces on a router, this method
   * solicits the actual IPv6 addresses of those interfaces.
   * @param numberOfInterfaces The number of interfaces on a particular router.
   * @param res Represents the servlet's response.
   */
  public void defineNumOfInterfaces(int numberOfInterfaces,
                                                   HttpServletResponse res){
    Calendar myCalendar;
    Date myDate;
    int host_part;
    String address;

    res.setContentType("text/html");

    try{
      PrintWriter out = res.getWriter();
      out.println("<HTML><HEAD><TITLE>Define a router's interfaces</TITLE>"
        + "<LINK REL=STYLESHEET TYPE=\"text/css\" HREF=\"/core-styles.css\">"
        + "</HEAD>");
      out.println("<BODY><H1>Define a router's interfaces</H1>");
      out.println("<H2>Enter the IPv6 addresses of these " + numberOfInterfaces
        + " interfaces: </H2>");
      out.println("<FORM METHOD=GET ACTION=\"/servlet/saam.server.Console\">"
        + "<INPUT TYPE=HIDDEN NAME=\"action\" VALUE=\"defineInterfaces\"><P>");
      myCalendar = Calendar.getInstance();
```

```
    myDate = myCalendar.getTime();
    host_part = myDate.getMinutes();
    for (int num = 1; num <= numberOfInterfaces; num++){
      address = "99.99.99." + num + ".0.0.0.0.0.0.0.0.0.0.0."+ host_part;
      out.println("<P class=\"component\">Interface #" + num + " address: "
        + "<INPUT TYPE=TEXT NAME=\"interface" + num + "\" VALUE=\"" + address
        +"\"></P>");
    }
    out.println("<INPUT TYPE=SUBMIT VALUE=Enter><P>");
    out.println("</FORM></BODY><HTML>");
  }catch(IOException ioe) {
    System.err.println(ioe);
  }
}

/**
 * Takes the IPv6 addresses that the user provided for each interface on the
 * router, instantiates a Router object and then passes it to myServer's
 * receive_LSA() method.
 * @param NumOfInterfacesToAdd Indicates how many IPv6 addresses are sent in
 * in the request.
 * @param req Represents the client's request.
 * @param res Represents the servlet's response.
 */
public void defineInterfaces(int NumOfInterfacesToAdd, HttpServletRequest req,
                                                    HttpServletResponse res){
  res.setContentType("text/html");

  String address;
  InterfaceID myInterface;
  Vector interfaces = new Vector();
  int bandwidth = 10000;
  Hello hello;

  try{
    PrintWriter out = res.getWriter();
    out.println("<HTML><HEAD><TITLE>Adding router to PIB</TITLE>"
      + "<LINK REL=STYLESHEET TYPE=\"text/css\" HREF=\"/core-styles.css\">"
      + "</HEAD>");
    out.println("<BODY><H1>The router has been added to PIB.</H1>");

    //add each interface
    for (int num = 1; num <= NumOfInterfacesToAdd; num++){
      address = req.getParameter("interface" + num);
      myInterface= new InterfaceID(IPv6Address.getByName(address),bandwidth);
      interfaces.addElement(myInterface);
    }
    hello = new Hello(interfaces);
    myServer.receive_Hello(hello);
    out.println("<FORM METHOD=GET ACTION=/servlet/saam.server.Console>"
      + "<INPUT TYPE=HIDDEN Name=\"action\" VALUE=\"console\">"
      + "<INPUT TYPE=SUBMIT VALUE=ReturnToConsole></FORM>");
    out.println("</FORM></BODY><HTML>");
  }catch(IOException ioe) {
    System.err.println(ioe);
  }
}

/**
 * Displays all of the routers and corresponding interfaces that are known
 * by the server's Path Information Base.
```

```java
 * @param res Represents the servlet's response.
 */
public void displayRouters(HttpServletResponse res){

  Vector routers = myServer.PIB.getAllRouterIds();
  int myRouter;
  Vector IPv6Addresses;
  IPv6Address myAddress;
  res.setContentType("text/html");

  try{
    PrintWriter out = res.getWriter();
    out.println("<HTML><HEAD><TITLE>Display All Routers</TITLE>"
      + "<LINK REL=STYLESHEET TYPE=\"text/css\" HREF=\"/core-styles.css\">"
      + "</HEAD>");
    out.println("<BODY><H1>Display All Routers</H1>");
    out.println("<FORM METHOD=GET ACTION=/servlet/saam.server.Console>"
      + "<INPUT TYPE=HIDDEN Name=\"action\" VALUE=\"console\">"
      + "<INPUT TYPE=SUBMIT VALUE=ReturnToConsole></FORM>");
    out.println("<H2>Here are the IPv6 addresses of these " + routers.size()
      + " routers: </H2>");
    for (int index = 0; index < routers.size(); index++){
      myRouter = ((Integer)routers.elementAt(index)).intValue();
      out.println("<H3>Router # " + myRouter + "</H3>");
      IPv6Addresses = myServer.PIB.getRouterInterfaces(myRouter);
      for (int num = 0; num < IPv6Addresses.size(); num++){
        myAddress = (IPv6Address)IPv6Addresses.elementAt(num);
        out.println("<P class=\"component\">Interface #" + num + " address: "
          + myAddress.toString() + "</P>");
      }
    }
    out.println("</FORM></BODY><HTML>");
  }catch(IOException ioe) {
    System.err.println(ioe);
  }
}

/**
 * Queries the user for the node_id of the router to be deleted.
 * @param res Represents the servlet's response.
 */
public void selectRouterForDeletion(HttpServletResponse res){
  res.setContentType("text/html");
  try{
    PrintWriter out = res.getWriter();
    out.println("<HTML><HEAD><TITLE>Select A Router For Deletion</TITLE>"
      + "<LINK REL=STYLESHEET TYPE=\"text/css\" HREF=\"/core-styles.css\">"
      + "</HEAD>");
    out.println("<BODY><H1>Delete A Router: </H1>");
    out.println("<FORM METHOD=GET ACTION=\"/servlet/saam.server.Console\">"
      + "<INPUT TYPE=HIDDEN NAME=\"action\" VALUE=\"deleteARouter\"><P>"
      + "<P class=\"component\">"
      + "What is the node_id of the router to be deleted?"
      + "<INPUT TYPE=TEXT NAME=\"router\"></P>"
      + "<INPUT TYPE=SUBMIT VALUE=Enter><P>");
    out.println("</FORM></BODY><HTML>");
  }catch(IOException ioe) {
    System.err.println(ioe);
  }
}
```

```java
/**
 * Takes the specified node id of the router to be deleted and performs the
 * deletion operation.
 * @param req Represents the client's request.
 * @param res Represents the servlet's response.
 */
public void deleteARouter(HttpServletRequest req, HttpServletResponse res){
  String node_id = req.getParameter("router");
  res.setContentType("text/html");
  try{
    PrintWriter out = res.getWriter();
    out.println("<HTML><HEAD><TITLE>Delete A Router</TITLE>"
      + "<LINK REL=STYLESHEET TYPE=\"text/css\" HREF=\"/core-styles.css\">"
      + "</HEAD>");
    myServer.PIB.deleteARouter((new Integer(node_id)).intValue());
    out.println("<BODY><H1>The router has been deleted.</H1>");
    out.println("<FORM METHOD=GET ACTION=/servlet/saam.server.Console>"
      + "<INPUT TYPE=HIDDEN Name=\"action\" VALUE=\"console\">"
      + "<INPUT TYPE=SUBMIT VALUE=ReturnToConsole></FORM>");
    out.println("</FORM></BODY><HTML>");
  }catch(IOException ioe) {
    System.err.println(ioe);
  }
}

/**
 * Queries the user for the link_id or subnet for which to search for
 * all attached routers.
 * @param res Represents the servlet's response.
 */
public void selectALink(HttpServletResponse res){
  Vector links;
  res.setContentType("text/html");
  try{
    PrintWriter out = res.getWriter();
    out.println("<HTML><HEAD><TITLE>Selection a Link</TITLE>"
      + "<LINK REL=STYLESHEET TYPE=\"text/css\" HREF=\"/core-styles.css\">"
      + "</HEAD>");
    out.println("<BODY><H1>Select a Link: </H1>");
    out.println("<H2>Currently defined subnets: ");
    links = myServer.PIB.getAllLinkIds();
    for (int index = 0; index < links.size(); index++){
      out.println("<P class=\"component\">" + links.elementAt(index)
        + "</P>");
    }
    out.println("<FORM METHOD=GET ACTION=\"/servlet/saam.server.Console\">"
      + "<INPUT TYPE=HIDDEN NAME=\"action\" VALUE=\"findRoutersOnLink\"><P>"
      + "<P class=\"component\">Which of these subnets should be searched?"
      + "<INPUT TYPE=TEXT NAME=\"link_id\"></P>"
      + "<INPUT TYPE=SUBMIT VALUE=Enter><P>");
    out.println("</FORM></BODY><HTML>");
  }catch(IOException ioe) {
    System.err.println(ioe);
  }
}

/**
 * Searches for and returns the routers that are attached to the specified
 * subnet link_id.
 * @param req Represents the client's request.
 * @param res Represents the servlet's response.
```

```
 */
public void findRoutersOnLink(HttpServletRequest req,
                                                HttpServletResponse res){
  Vector routers = new Vector();
  int myRouter;
  Vector IPv6Addresses = new Vector();
  IPv6Address myAddress;
  IPv6Address link_id = null;

  res.setContentType("text/html");
  try{
  link_id = IPv6Address.getByName(req.getParameter("link_id"));
  }catch(UnknownHostException uhe){
    System.out.println("Console:findRoutersOnLink: " + uhe);
  }
  try{
    PrintWriter out = res.getWriter();
    out.println("<HTML><HEAD><TITLE>Find Routers on a Link</TITLE>"
      + "<LINK REL=STYLESHEET TYPE=\"text/css\" HREF=\"/core-styles.css\">"
      + "</HEAD>");
    routers = myServer.PIB.findRoutersOnLink(link_id);
    out.println("<BODY><H1>These " + routers.size()
      + " routers are attached to subnet: "
      + link_id + "</H1>");
    out.println("<FORM METHOD=GET ACTION=/servlet/saam.server.Console>"
      + "<INPUT TYPE=HIDDEN Name=\"action\" VALUE=\"console\">"
      + "<INPUT TYPE=SUBMIT VALUE=ReturnToConsole></FORM>");
    for (int index = 0; index < routers.size(); index++){
      myRouter = ((Integer)routers.elementAt(index)).intValue();
      out.println("<H3>Router # " + myRouter + "</H3>");
      IPv6Addresses = myServer.PIB.getRouterInterfaces(myRouter);
      for (int num = 0; num < IPv6Addresses.size(); num++){
        myAddress = (IPv6Address)IPv6Addresses.elementAt(num);
        out.println("<P class=\"component\">Interface #" + num + " address: "
          + myAddress.toString() + "</P>");
      }
    }
    out.println("</FORM></BODY><HTML>");
  }catch(IOException ioe) {
    System.err.println(ioe);
  }
}

/**
 * The web server invokes this method each time it receives a POST
 * request for this servlet.
 * @param req Represents the client's request.
 * @param res Represents the servlet's response.
 */
public void doPost(HttpServletRequest req, HttpServletResponse res)
                        throws ServletException, IOException {

  if (req.getMethod().equals("HEAD")) return;
  res.setContentType("text/html");
  PrintWriter out = res.getWriter();

  String action = req.getParameter("action");

  if (action.equals("INSERT")) {
    }
```

```
    else if (action.equals("DELETE")) {
      }

  }

}
```

```
package saam.server;

import saam.net.*;
import saam.message.*;
import saam.util.*;
import java.net.*;
import java.sql.*;
import java.util.*;
import java.io.*;

/**
 * The <em>DatabaseStructure</em> is a Path Information Base object
within the
 * SAAM architecture that performs SQL queries on the database
containing the
 * information needed to obtain a picture of the network for use in
assigning
 * flows to paths.
 */
public class DatabaseStructure extends PathInformationBase{

  /** A connection to a given database. */
  private Connection con = null;

  /** A boolean that will allow the showing of comments. */
  private boolean showComments = false;

  private SAAMRouterGui gui;

//**********************************************************************
*******
// These methods handle database connection details.
//**********************************************************************
******/

  /**
   * Constructs a DatabaseStructure object that will connect to an
Oracle
   * database using Sun's JDBC-ODBC bridge.
   */
  public DatabaseStructure(){

    gui=new SAAMRouterGui("PIB");
    /** Sun's JDBC-ODBC bridge driver */
    String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
    /**
     * A JDBC URL identifies an individual database in a driver-specific
manner.
     */
    String url = "jdbc:odbc:PIB";
    /** The username for the database. */
    String username = "PIB";
    /** The database user's password. */
    String password = "PIB";

    try {
      // Load (and therefore register) the db driver
      Class.forName(driver);
      // Ask DriverManger to etablish a connection to the db
      con = DriverManager.getConnection(url, username, password);
    }
```

```
    catch (ClassNotFoundException e) {
      gui.sendText("PIB: Couldn't load database driver: " + e);
    }
    catch (SQLException e) {
      gui.sendText("PIB: Couldn't get db connection: " + e);
    }
    //if (showComments) {
      gui.sendText("PIB: DatabaseStructure: Connection established.");
    //}
  }

  /**
   * Removes all current path data from the database.. Its most commonly
used
   * during initialization of a SAAM server for a new network.
   */
  public void deleteAllData() {

    /** Used for passing SQL statements to the database. */
    Statement statement;

    try {
      statement = con.createStatement();
      statement.executeUpdate("DELETE FROM FLOW;");
      statement.executeUpdate("DELETE FROM INTERFACES;");
      statement.executeUpdate("DELETE FROM LINK;");
      statement.executeUpdate("DELETE FROM PATH;");
      statement.executeUpdate("DELETE FROM ROUTER;");
      statement.executeUpdate("DELETE FROM SERVICE_LEVEL_PIPE;");
      statement.executeUpdate("DELETE FROM SLP_SEQUENCE;");
      statement.executeUpdate("DELETE FROM TRAVERSES;");
      statement.close();
    } catch(SQLException e) {
      gui.sendText("PIB: deleteAllData: " + e);
    }
    if (showComments) {
      gui.sendText("PIB: deleteAllData: All data deleted.");
    }
  }

  /* Standard code for db methods...
    Statement statement;
    ResultSet rs;
    try{
      statement = con.createStatement();
      statement.close();
    }catch(SQLException e){
      gui.sendText("PIB: receive_LSA: SQLException: " + e);
    }
  */

//***********************************************************************
*******
// These methods are used to process link state advertisements from
routers
//***********************************************************************
******/

  /**
   * Determines whether a given router exists yet within the PIB.
```

```
    * @param IPv6Addresses A vector of interface addresses contained
 within
    * a Hello or an LSA message.
    * @returns node_id The id of the node containing at least one of the
    * interface addresses in the vector that was passed.
    */
   public int doesRouterExist(Vector IPv6Addresses){
     /** Used for passing SQL statements to the database. */
     Statement statement;
     /** Used to store result sets from database queries. */
     ResultSet rs;
     String addresses;
     IPv6Address myIPv6Address;
     int node_id = 0;
     try{

       addresses = "";
       for (int i = 0; i < IPv6Addresses.size(); i++) {
         myIPv6Address = (IPv6Address)IPv6Addresses.elementAt(i);
         addresses = addresses.concat(myIPv6Address.toString());
         if (i < (IPv6Addresses.size()-1)){
           addresses = addresses.concat("', '");
         }
       }

       statement = con.createStatement();
       // search for existing interfaces
       rs = statement.executeQuery("SELECT * FROM INTERFACES WHERE
ADDRESS IN ('"
         + addresses + "');");
       // if any are found, then return true
       if (rs.next()){
         node_id = (new Integer(rs.getString("NODE_ID"))).intValue();
       }
       statement.close();
     }catch(SQLException e){
       gui.sendText("PIB: doesRouterExist: SQLException: " + e);
     }
     if (showComments) {
       if (node_id != 0)
         gui.sendText("PIB: doesRouterExist: Router "
           + node_id + " exists.");
       else
         gui.sendText("PIB: doesRouterExist: Router is not in
database.");
     }
     return node_id;
   }

   /**
    * Finds an unassigned node id and adds it to the PIB. It is commonly
used
    * for assigning a new node_id to a previously unknown router.
    * @returns max_node_id An unassigned router id.
    */
   public int getNewNodeId(){

     Statement statement;
     ResultSet rs;
     String node_id;
     int max_node_id = 0;
```

```
    try{
      statement = con.createStatement();
      // find the largest assigned node_id
      rs = statement.executeQuery("SELECT NODE_ID FROM ROUTER;");
      while (rs.next()){
        node_id = rs.getString("NODE_ID");
        if (max_node_id < (new Integer(node_id)).intValue())
          max_node_id = (new Integer(node_id)).intValue();
      }
      // increment it and then assign that id to the new router
      max_node_id++;
      statement.executeUpdate("INSERT INTO ROUTER (NODE_ID) VALUES ("
        + (new Integer(max_node_id)).toString() + ");");
      statement.close();
    }catch(SQLException e){
      gui.sendText("PIB: assignNewNodeId: SQLException: " + e);
    }
    if (showComments) {
      gui.sendText("PIB: assignNewNodeId: Router's id assigned: "
        + max_node_id);
    }
    return max_node_id;
  }

  /**
   * Determines whether a given interface exists yet within the PIB.
   * @param myIPv6Address The interface address contained within a hello
or LSA
   * message.
   * @returns found True if the interface address already exists within
the PIB.
   */
  public boolean doesInterfaceExist(IPv6Address myIPv6Address){
    Statement statement;
    ResultSet rs;
    boolean found = false;
    try{
      statement = con.createStatement();
      rs = statement.executeQuery("SELECT * FROM INTERFACES WHERE
ADDRESS = '"
        + myIPv6Address.toString() + "';");
      // if any are found, then return true
      if (rs.next())
        found = true;
      statement.close();
    }catch(SQLException e){
      gui.sendText("PIB: doesInterfaceExist: SQLException: " + e);
    }
    if (showComments) {
      if (found)
        gui.sendText("PIB: doesInterfaceExist: Interface "
          + myIPv6Address.toString() + " is found.");
      else
        gui.sendText("PIB: doesInterfaceExist: Interface "
          + myIPv6Address.toString() + " is not found.");
    }
    return found;
  }

  /**
   * Determines whether a given link exists yet within the PIB.
```

```
    * @param address The IPv6 address of an interface.
    * @returns found True if the link address already exists within the
PIB.
    */
   public boolean doesLinkExist(IPv6Address address){
     Statement statement;
     ResultSet rs;
     boolean found = false;
     try{
       statement = con.createStatement();
       rs = statement.executeQuery("SELECT * FROM LINK WHERE LINK_ID = '"
         + address.getNetworkAddress() + "';" );
       // if any are found, then return true
       if (rs.next())
         found = true;
     statement.close();
     }catch(SQLException e){
       gui.sendText("PIB: doesLinkExist: SQLException: " + e);
     }
     if (showComments) {
       if (found)
         gui.sendText("PIB: doesLinkExist: Link "
           + address.getNetworkAddress() + " is found.");
       else
         gui.sendText("PIB: doesLinkExist: Link "
           + address.getNetworkAddress() + " is not found.");
     }
     return found;
   }

   /**
    * Adds a new link to the PIB.
    * @param address The IPv6 address of an interface.
    * @param max_bandwidth The max transmission rate over this network
segment.
    */
   public void addLink(IPv6Address address, int max_bandwidth){
     Statement statement;
     ResultSet rs;
     try{
       statement = con.createStatement();
       statement.executeUpdate("INSERT INTO LINK VALUES ('"
         + address.getNetworkAddress() + "', '" + max_bandwidth + "');");
     statement.close();
     }catch(SQLException e){
       gui.sendText("PIB: addLink: SQLException: " + e);
     }
     if (showComments) {
       gui.sendText("PIB: addLink: Link " + address.getNetworkAddress()
         + " is added.");
     }
   }

   /**
    * Adds a new interface to the PIB.
    * @param node_id The id of the router whose interface is being added.
    * @param address The IPv6 address of an interface.
    */
   public void addInterface(int node_id, IPv6Address address){
     Statement statement;
     ResultSet rs;
```

```
    try{
      statement = con.createStatement();
      statement.executeUpdate("INSERT INTO INTERFACES VALUES ('"
        + node_id + "', '" + address.getNetworkAddress() + "', '"
        + address + "');");
    statement.close();
    }catch(SQLException e){
      gui.sendText("PIB: addInterface: SQLException: " + e);
    }
    if (showComments) {
      gui.sendText("PIB: addInterface: node_id = " + node_id + ",
link_id = "
        + address.getNetworkAddress() + ", address = " + address + " is
added.");
    }
  }

  /**
   * Determines whether a service level pipe exists yet within the PIB.
   * @param address The IPv6 address of an interface.
   * @param service_level The level of service that this logical pipe is
   * providing.
   * @returns found True if this SLP is already in the PIB.
   */
  public boolean doesSLPExist(IPv6Address myIPv6Address, int
service_level){
    Statement statement;
    ResultSet rs;
    boolean found = false;
    String node_id = null;
    try{
      statement = con.createStatement();
      rs = statement.executeQuery("SELECT * FROM INTERFACES WHERE
ADDRESS = '"
        + myIPv6Address.toString() + "';");
      if (rs.next()){
        node_id = rs.getString("NODE_ID");
      }
      rs = statement.executeQuery(
        "SELECT * FROM SERVICE_LEVEL_PIPE WHERE LINK_ID = '"
        + myIPv6Address.getNetworkAddress().toString() + "' AND
SOURCE_ROUTER = '"
        + node_id + "' AND SLP_ID = '" + service_level + "';");
       // if any are found, then return true
      if (rs.next())
        found = true;
      statement.close();
    }catch(SQLException e){
      gui.sendText("PIB: doesSLPExist: SQLException: " + e);
    }
    if (showComments) {
      if (found)
        gui.sendText("PIB: doesSLPExist: SLP from router "
          + node_id + " is found.");
      else
        gui.sendText("PIB: doesSLPExist: SLP from router "
          + node_id + " is not found.");
    }
    return found;
  }
```

```
  /**
   * Updates the status of a known SLP's delay, loss_rate, and
throughput.
   * @param address The IPv6 address of an interface.
   * @param service_level The level of service that this logical pipe is
   * providing.
   * @param delay The average delay experienced by a packet's stay in
the
   * particular SLP outbound queue.
   * @param loss_rate The average loss_rate experienced by packets in a
   * particular SLP.
   * @param throughput The average throughput provided by a particular
SLP.
   */
  public void updateSLP(IPv6Address address, int service_level, int
delay,
      int loss_rate, int throughput){
      Statement statement;
      ResultSet rs;
      String node_id = null;
      try{
        statement = con.createStatement();
        rs = statement.executeQuery("SELECT * FROM INTERFACES WHERE "
          + "ADDRESS = '" + address.toString() + "';");
        if (rs.next()){
          node_id = rs.getString("NODE_ID");
        }
        statement.executeUpdate(
          "UPDATE SERVICE_LEVEL_PIPE SET OBSERVED_DELAY = '" + delay
           + "', OBSERVED_LOSS_RATE = '" + loss_rate + "',
OBSERVED_THROUGHPUT = '"
           + throughput + "' WHERE SLP_ID = '" + service_level + "' AND
SOURCE_ROUTER = '"
           + node_id + "' AND LINK_ID = '" + address.getNetworkAddress() +
"';");
        statement.close();
      }catch(SQLException e){
        gui.sendText("PIB: updateSLP: SQLException: " + e);
      }
      if (showComments) {
        gui.sendText("PIB: updateSLP: SLP " + service_level + " is
updated.");
      }
  }

  /**
   * Adds a previously unknown SLP to the PIB along with its targeted
QoS.
   * @param address The IPv6 address of an interface.
   * @param service_level The level of service that this logical pipe is
   * providing.
   * @param target_delay The average delay experienced by a packet's
stay in the
   * particular SLP outbound queue.
   * @param target_loss_rate The average loss_rate experienced by
packets in a
   * particular SLP.
   * @param target_throughput The average throughput provided by a
particular SLP.
   */
```

```
   public void addSLP(IPv6Address address, int service_level, int
 target_delay,
     int target_loss_rate, int target_throughput){
     Statement statement;
     ResultSet rs;
     String node_id = null;
     String delay = "0", loss_rate = "0", throughput = "0";
     try{
       statement = con.createStatement();
       rs = statement.executeQuery("SELECT * FROM INTERFACES WHERE "
         + "ADDRESS = '" + address.toString() + "';");
       while (rs.next()){
         node_id = rs.getString("NODE_ID");
       }
       statement.executeUpdate("INSERT INTO SERVICE_LEVEL_PIPE (SLP_ID,"
         + " SOURCE_ROUTER, LINK_ID, TARGET_DELAY, TARGET_LOSS_RATE,
TARGET_THROUGHPUT,"
         + "OBSERVED_DELAY, OBSERVED_LOSS_RATE, "
         + "OBSERVED_THROUGHPUT) VALUES ('" + service_level + "', '"
         + node_id + "', '" + address.getNetworkAddress().toString() +
"', '"
         + target_delay + "', '" + target_loss_rate + "', '" +
target_throughput + "', '"
         + delay + "', '" + loss_rate + "', '" + throughput + "');");
       statement.close();
     }catch(SQLException e){
       gui.sendText("PIB: addSLP: SQLException: " + e);
     }
     if (showComments) {
       gui.sendText("PIB: addSLP: SLP " + service_level + " is added to
"+address);
     }
   }

//*******************************************************************
*******
// These methods are used to process a flow request from a host
//*******************************************************************
******/

   /**
    * Finds a router id that has an interface to on the same link as the
host
    * making a flow request.
    * @param address The IPv6 address of the interface of the host
requesting
    * the flow.
    * @returns ARouter The router id of the first router found on this
link.
    */
  public int findARouterOnLink(IPv6Address address){
    Statement statement1,statement2;
    ResultSet rs1,rs2;
    int ARouter = 0;
    try{
      statement1 = con.createStatement();
      statement2 = con.createStatement();
      // check to see if requesting host is a router itself
      rs1 = statement1.executeQuery("SELECT * FROM INTERFACES WHERE
ADDRESS = '"
        + address.toString() + "';");
```

```
        // if so, return its node_id
        if (rs1.next()){
          ARouter = (new Integer(rs1.getString("NODE_ID"))).intValue();
        }
        // otherwise, find any other router on the same subnet
        else {
          rs2 = statement2.executeQuery("SELECT * FROM INTERFACES WHERE
LINK_ID = '"
            + address.getNetworkAddress() + "';");
          if (rs2.next()){
            ARouter = (new Integer(rs2.getString("NODE_ID"))).intValue();
          }
        }
        statement1.close();
        statement2.close();
      }catch(SQLException e){
        gui.sendText("PIB: findARouterOnLink: SQLException: " + e);
      }
      if (showComments) {
        gui.sendText("PIB: findARouterOnLink: Router "
          + ARouter + " on " + address.getNetworkAddress()+ " is assigned
for "
          + address);
      }
      return ARouter;
    }

    /**
     * Determines if there is a path that can support a particular flow
request.
     * A value of zero is returned if no path can support this QoS.
     * @param source_router The node id of a router on the same physical
link as
     * the source host.
     * @param destination_router The node id of a router on the same
physical
     * link as the destination host.
     * @param myFlowRequest A host's request for the establishment of a
flow.
     * @returns path_id The id of a path that can support this request.
     */
    public int getPathThatCanSupportFlowRequest(int source_router,
                             int destination_router, FlowRequest
myFlowRequest){
      Statement statement1,statement2;
      ResultSet rs1,rs2;
      String max_throughput;
      String delay = null;
      String loss_rate = null;
      String throughput = null;
      int path_id = 0;
      try{
        statement1 = con.createStatement();
        statement2 = con.createStatement();
        // find a path between these two routers that can meet the QoS
request
        rs1 = statement1.executeQuery("SELECT MAX(EFFECTIVE_THROUGHPUT)
MAXT FROM PATH "
          + "WHERE SOURCE_ROUTER = '" + source_router
          + "' AND DESTINATION_ROUTER = '" + destination_router
```

```
        + "' AND EFFECTIVE_DELAY <= '" +
myFlowRequest.getRequestedDelay()
        + "' AND EFFECTIVE_LOSS_RATE <= '"
        + myFlowRequest.getRequestedLossRate()
        + "' AND EFFECTIVE_THROUGHPUT >= '"
        + myFlowRequest.getRequestedThroughput() + "';");
      if (rs1.next()){
        max_throughput = rs1.getString("MAXT");
        rs2 = statement2.executeQuery("SELECT * FROM PATH "
          + "WHERE SOURCE_ROUTER = '" + source_router
          + "' AND DESTINATION_ROUTER = '"
          + destination_router
          + "' AND EFFECTIVE_DELAY <= '" +
myFlowRequest.getRequestedDelay()
          + "' AND EFFECTIVE_LOSS_RATE <= '"
          + myFlowRequest.getRequestedLossRate()
          + "' AND EFFECTIVE_THROUGHPUT = '" + max_throughput + "';");
        if (rs2.next()){
          path_id = (new Integer(rs2.getString("PATH_ID"))).intValue();
          delay = rs2.getString("EFFECTIVE_DELAY");
          loss_rate = rs2.getString("EFFECTIVE_LOSS_RATE");
          throughput = rs2.getString("EFFECTIVE_THROUGHPUT");
        }
      }
      statement1.close();
      statement2.close();
    }catch(SQLException e){
      gui.sendText("PIB: getPathThatCanSupportFlowRequest: SQLException:
" + e);
    }
    if (showComments) {
      gui.sendText("PIB: getPathThatCanSupportFlowRequest:"
        + " SELECT MAX(EFFECTIVE_THROUGHPUT) MAXT FROM PATH "
        + "WHERE SOURCE_ROUTER = '" + source_router
        + "' AND DESTINATION_ROUTER = '" + destination_router + "'");
      gui.sendText("        AND EFFECTIVE_DELAY <= '"
        + myFlowRequest.getRequestedDelay()
        + "' AND EFFECTIVE_LOSS_RATE <= '"
        + myFlowRequest.getRequestedLossRate()
        + "' AND EFFECTIVE_THROUGHPUT >= '"
        + myFlowRequest.getRequestedThroughput() + "';");
      if (path_id != 0)
        gui.sendText("PIB: getPathThatCanSupportFlowRequest: path
assigned = "
          + path_id + " with effective D" + " = " + delay
          + "ms, LR = " + loss_rate + "%, T = " + throughput
          + "kbps");
      else{
        gui.sendText("PIB: getPathThatCanSupportFlowRequest: "
          + "Flow request cannot be supported.");
      }
    }
    return path_id;
  }

  /**
   * Finds an unassigned flow id and assigns this new flow to a path.
   * @param path_id The id of a path that can support this request.
   * @param source_router The node id of a router on the same physical
link as
   * the source host.
```

```
   * @param destination_router The node id of a router on the same
physical
   * link as the destination host.
   * @param myFlowRequest A host's request for the establishment of a
flow.
   * @returns max_flow_id The id that is being assigned to this flow.
   */
  public int getNewFlowId(int path_id, int source_router,
                              int destination_router, FlowRequest
myFlowRequest){
    Statement statement;
    ResultSet rs;
    String flow_id;
    /** The max flow id assigned so far. */
    int max_flow_id = 0;
    try{
      statement = con.createStatement();
      rs = statement.executeQuery("SELECT FLOW_ID FROM FLOW;");
      while (rs.next()){
        flow_id = rs.getString("FLOW_ID");
        if (max_flow_id < (new Integer(flow_id)).intValue())
          max_flow_id = (new Integer(flow_id)).intValue();
      }
      max_flow_id++;
      // increment it and assign it along with the QoS parameters
      statement.executeUpdate("INSERT INTO FLOW (FLOW_ID, PATH_ID,
SOURCE_NODE,"
        + " DESTINATION_NODE, NEGOTIATED_DELAY, NEGOTIATED_LOSS_RATE, "
        + "NEGOTIATED_THROUGHPUT) VALUES ("
        +  max_flow_id + ", " + path_id + ", "
        + source_router + ", "
        + destination_router + ", "
        + myFlowRequest.getRequestedDelay() + ", "
        + myFlowRequest.getRequestedLossRate() + ", "
        + myFlowRequest.getRequestedThroughput() + ");");
      statement.close();
    }catch(SQLException e){
      gui.sendText("PIB: getNewFlowId: SQLException: " + e);
    }
    if (showComments) {
      gui.sendText("PIB: getNewFlowId: Flow "
        + max_flow_id + " assigned.");
    }
    return max_flow_id;
  }

  /**
   * Retrieves the sequence of SLP that make up a given path.
   * @param path_id The id of the path in question.
   * @returns slps_in_path A vector of SLPs that compose this path.
   */
  public Vector getSLPSequenceOfPath(int path_id){
    Statement statement;
    ResultSet rs;
    Vector slps_in_path = new Vector();
    SLPSequence SLP_sequence;
    try{
      statement = con.createStatement();
      // get the sequence of service level pipes that compose the path
      rs = statement.executeQuery("SELECT * FROM SLP_SEQUENCE WHERE
PATH_ID ='"
```

```
        + path_id + "' ORDER BY SLP_SEQUENCE ASC;");
      while(rs.next()){
        SLP_sequence = new SLPSequence();
        try{
SLP_sequence.setLinkId(IPv6Address.getByName(rs.getString("LINK_ID")));
        }
        catch(UnknownHostException uhe){
          gui.sendText("PIB: getSLPSequenceOfPath: UnknownHostException:
"
            + uhe);
        }
        SLP_sequence.setServiceLevel((new Integer(rs.getString
("SLP_ID"))).intValue());
        SLP_sequence.setSourceRouter((new Integer(rs.getString
("SOURCE_ROUTER"))).intValue());
        SLP_sequence.setPathId((new Integer(rs.getString
("PATH_ID"))).intValue());
        SLP_sequence.setSequenceNumber((new Integer(rs.getString
("SLP_SEQUENCE"))).intValue());
        slps_in_path.addElement(SLP_sequence);
      }
      statement.close();
    }catch(SQLException e){
      gui.sendText("PIB: getSLPSequenceOfPath: SQLException: " + e);
    }
    if (showComments) {
      gui.sendText("PIB: getSLPSequenceOfPath: SLP sequence of " +
path_id
        + " of size " + slps_in_path.size() + " is retrieved.");
    }
    return slps_in_path;
  }

  /**
   * Retrieves  the IPv6 address of an interface.
   * @param node_id The id of the router whose interface is being
queried.
   * @param link_id The network portion of the IPv6 address of an
interface.
   * @returns address The address of the interface that connects this
node and
   * link.
   */
  public IPv6Address getInterfaceAddress(int node_id, IPv6Address
link_id){
    Statement statement;
    ResultSet rs;
    IPv6Address address = new IPv6Address();
    try{
      statement = con.createStatement();
      rs = statement.executeQuery("SELECT ADDRESS FROM INTERFACES WHERE
"
        + "NODE_ID = '" + node_id + "' AND LINK_ID = '" +
link_id.toString() + "';");
      if(rs.next())
        address = IPv6Address.getByName(rs.getString("ADDRESS"));
```

```
        statement.close();
      }catch(SQLException e){
        gui.sendText("PIB: getInterfaceAddress: SQLException: " + e);
      }
      catch(UnknownHostException uhe){
        gui.sendText("PIB: getInterfaceAddress:UnknownHostException: "
          + uhe);
      }
      if (showComments) {
        gui.sendText("PIB: getInterfaceAddress: Interface address "
          + address + " returned.");
      }
      return address;
    }

//**********************************************************************
*******
// These methods are used to determine all possible paths across the
network
//**********************************************************************
******/

    /**
     * Retrieve all of the router ids assigned by the PIB so far.
     * @returns V A vector of all assigned router ids.
     */
    public Vector getAllRouterIds(){
      Statement statement;
      ResultSet rs;
      Vector V = new Vector();
      try{
        statement = con.createStatement();
        //search for existing routers
        rs = statement.executeQuery("SELECT * FROM ROUTER;");

        // if any are found, then add them to V
        while (rs.next()) {
          V.addElement(new Integer(rs.getInt("NODE_ID")));
        }
        statement.close();
      }catch(SQLException e){
        gui.sendText("PIB: getAllRouterIds: SQLException: " + e);
      }
      if (showComments) {
        gui.sendText("PIB: getAllRouterIds: routers in PIB:");
        gui.sendText(V.toString());
      }
      return V;
    }

    /**
     * Retrieves the maximum service level of this SAAM region.
     * @returns max_slp_id The numerically highest service level id
assigned.
     */
    public int findMaxServiceLevel(){
      Statement statement;
      ResultSet rs;
      String slp_id;
      int max_slp_id = 0;
```

```
    try{
      statement = con.createStatement();
      rs = statement.executeQuery("SELECT SLP_ID FROM
SERVICE_LEVEL_PIPE;");
      while (rs.next()){
        slp_id = rs.getString("SLP_ID");
        if (max_slp_id < (new Integer(slp_id)).intValue())
          max_slp_id = (new Integer(slp_id)).intValue();
      }
      statement.close();
    }catch(SQLException e){
      gui.sendText("PIB: findMaxServiceLevel: SQLException: " + e);
    }
    if (showComments) {
      gui.sendText("PIB: findMaxServiceLevel: The max service level is "
        + (new Integer(max_slp_id)).toString());
    }
    return max_slp_id;
  }

  /**
   * Retrieves an array of parents for each router.  A parent is a
directly
   * connected node.
   * @param V A vector of all router ids.
   * @param service_level The level of service for which paths are being
built
   * for.
   * @returns parent A hashtable of vectors containing the parents of
each router.
   */
  public Hashtable getParents(Vector V, int service_level){
    Statement statement1,statement2;
    ResultSet rs1,rs2;
    Vector myVector = new Vector();
    Hashtable parent = new Hashtable();
    String link_id = null;
    Integer node_id;
    try{
      statement1 = con.createStatement();
      statement2 = con.createStatement();
      // for each "destination" node
      for (int index = 0; index < V.size(); index++){
        myVector = new Vector();
        // get all directly connected links
        rs1 = statement1.executeQuery("SELECT LINK_ID FROM INTERFACES
WHERE "
          + " NODE_ID = '" + V.elementAt(index) + "';");
        // for each directly connected link
        while (rs1.next()){
          link_id = rs1.getString("LINK_ID");
          //get all nodes that that are connected to the link
          rs2 = statement2.executeQuery("SELECT NODE_ID FROM INTERFACES
WHERE "
            + " LINK_ID = '" + link_id + "';");
          //for each connected node
          while (rs2.next()){
            node_id = new Integer(rs2.getString("NODE_ID"));
            // if this node is not the "destination" node
            if (!node_id.equals(V.elementAt(index))) {
              // add it to the parent vector of this "destination" node
```

```
              myVector.addElement(node_id);
            } //end if
          } // end while rs2
        } //end while rs1
        parent.put(V.elementAt(index),myVector);
      } //end for each destination node
      statement1.close();
      statement2.close();
    }catch(SQLException e){
      gui.sendText("PIB: getParents: SQLException: " + e);
    }
    if (showComments) {
        gui.sendText("PIB: getParents: Parent hashtable:");
        gui.sendText(parent.toString());
    }
    return parent;  .
  }

  /**
   * Assigns a path from a source and to a destination.
   * @param source_router The node_id of the source of the path.
   * @param destination_router The node_id of the destination of the
path.
   * @returns max_path_id The id to be assigned to this new path.
   */
  public int getNewPathId(int source_router,int destination_router){
    Statement statement;
    ResultSet rs;
    String path_id;
    int max_path_id = 0;
    try{
      statement = con.createStatement();
      rs = statement.executeQuery("SELECT PATH_ID FROM PATH;");
      while (rs.next()){
        path_id = rs.getString("PATH_ID");
        if (max_path_id < (new Integer(path_id)).intValue())
          max_path_id = (new Integer(path_id)).intValue();
      }
      max_path_id++;
      statement.executeUpdate("INSERT INTO PATH (PATH_ID,SOURCE_ROUTER,"
        +"DESTINATION_ROUTER) VALUES(" + max_path_id + ","
        + source_router + "," + destination_router + ");");
      statement.close();
    }catch(SQLException e){
      gui.sendText("PIB: getNewPathId: SQLException: " + e);
    }
    if (showComments) {
      gui.sendText("PIB: getNewPathId: Path " + max_path_id + " from "
        +source_router+" to "+destination_router+" is inserted.");
    }
    return max_path_id;
  }

  /**
   * Identifies the id of the physical link between two adjacent
routers. If no
   * link exists between a source and destination router, the default
address
   * of all zeros is returned.
   * @param source_router The node_id of a router.
   * @param destination_router The node_id of an adjacent router.
```

```
   * @returns subnet The link id between this source and destination.
   */
  public IPv6Address getLinkBetween(int source_router, int
destination_router){
    Statement statement;
    ResultSet rs;
    String link_id;
    Vector link_ids = new Vector();
    IPv6Address subnet = new IPv6Address();
    try{
      statement = con.createStatement();
      rs = statement.executeQuery("SELECT LINK_ID FROM INTERFACES WHERE
"
        + "NODE_ID = '" + source_router + "';");
      while (rs.next()){
        link_id = rs.getString("LINK_ID");
        link_ids.addElement(link_id);
      }
      rs = statement.executeQuery("SELECT LINK_ID FROM INTERFACES WHERE
"
        + "NODE_ID = '" + destination_router + "';");
      while (rs.next()){
        link_id = rs.getString("LINK_ID");
        if (link_ids.contains(link_id)){
          subnet = IPv6Address.getByName(link_id);
        }
      }
      statement.close();
    }catch(SQLException e){
      gui.sendText("PIB: getLinkBetween: SQLException: " + e);
    }
    catch(UnknownHostException uhe) {
      gui.sendText("PIB: getLinkBetween: " + uhe);
    }
    if (showComments) {
      gui.sendText("PIB: getLinkBetween: Link between " + source_router
+ " and "
        + destination_router + " is " + subnet);
    }
    return subnet;
  }

  /**
   * Assigns a service level pipe sequence entry in the building of a
path.
   * @param service_level The level of service for which paths are being
built
   * for.
   * @param source_router The node_id of the source of the SLP.
   * @param link_id The subnet that this SLP goes over.
   * @param path_id The id assigned to the path.
   * @param sequence_number The number assigned to specify the sequence
of this
   * SLP in the path.
   */
  public void assignSLPSequence(int service_level, int source_router,
    IPv6Address link_id, int path_id, int sequence_number){
    Statement statement;
    ResultSet rs;
    try{
      statement = con.createStatement();
```

```
      statement.executeUpdate("INSERT INTO SLP_SEQUENCE (SLP_ID,"
        + "SOURCE_ROUTER,LINK_ID,PATH_ID,SLP_SEQUENCE) VALUES('"
        + service_level + "', '" + source_router
        + "', '" + link_id.toString() + "', '" + path_id + "', '"
        + sequence_number + "');");
      statement.close();
    }catch(SQLException e){
      gui.sendText("PIB: assignSLPSequence: SQLException: " + e);
    }
    if (showComments) {
      gui.sendText("PIB: assignSLPSequence: SLP from "+source_router
        +" to link " + link_id + " on path "+ path_id
        + " is assigned sequence number "+sequence_number);
    }
  }

//**********************************************************************
*******
// These methods are used to determine the effective QoS of paths
//**********************************************************************
******/

  /**
   * Retrieves a vector of all path ids constructed by the PIB.
   * @returns path_ids All of the path ids known to the PIB.
   */
  public Vector getAllPathIds(){
    Statement statement;
    ResultSet rs;
    Vector path_ids = new Vector();
    Integer path_id;
    try{
      statement = con.createStatement();
      rs = statement.executeQuery("SELECT PATH_ID FROM PATH;");
      while (rs.next()){
        path_id = new Integer(rs.getString("PATH_ID"));
        path_ids.addElement(path_id);
      }
      statement.close();
    }catch(SQLException e){
      gui.sendText("PIB: getAllPathIds: SQLException: " + e);
    }
    if (showComments) {
      gui.sendText("PIB: getAllPathIds: paths in the PIB:");
      gui.sendText(path_ids.toString());
    }
    return path_ids;
  }

  /**
   * Retrieves the SLPs that make up a given path.
   * @param path_id The id of the path in question.
   * @returns slps_in_path The SLPs that make up the path.
   */
  public Vector getSLPsOfPath(int path_id){
    Statement statement1,statement2,statement3;
    ResultSet rs1,rs2,rs3;
    Vector slps_in_path = new Vector();
    SLP mySLP;
    String service_level, link_id, source_router,observed_delay,
      observed_loss_rate,observed_utilization,target_throughput;
```

```
    int observed_throughput;
    try{
      statement1 = con.createStatement();
      statement2 = con.createStatement();
      statement3 = con.createStatement();
      // get the sequence of service level pipes that compose the path
      rs1 = statement1.executeQuery("SELECT * FROM SLP_SEQUENCE WHERE
PATH_ID ='"
        + path_id + "';");
      while(rs1.next()){
        mySLP = new SLP();
        link_id = rs1.getString("LINK_ID");
        service_level = rs1.getString("SLP_ID");
        mySLP.setServiceLevel((new Integer(service_level)).intValue());
        source_router = rs1.getString("SOURCE_ROUTER");
        rs2 = statement2.executeQuery("SELECT ADDRESS FROM INTERFACES
WHERE LINK_ID = '"
          + link_id + "' AND NODE_ID = '" + source_router + "';");
        if (rs2.next()){
          try{
mySLP.setAddress(IPv6Address.getByName(rs2.getString("ADDRESS")));
          }catch(UnknownHostException uhe){
            gui.sendText("PIB: getRouterInterfaces:UnknownHostException:
"
              + uhe);
          }
        }
        rs3 = statement3.executeQuery("SELECT * FROM SERVICE_LEVEL_PIPE
WHERE SLP_ID = '"
          + service_level + "' AND SOURCE_ROUTER = '"+ source_router +
"' AND LINK_ID = '"
          + link_id + "';");
        if (rs3.next()){
          target_throughput = rs3.getString("TARGET_THROUGHPUT");
          mySLP.setTargetThroughput((new
Integer(target_throughput)).intValue());
          observed_delay = rs3.getString("OBSERVED_DELAY");
          mySLP.setDelay((new Integer(observed_delay)).intValue());
          observed_loss_rate = rs3.getString("OBSERVED_LOSS_RATE");
          mySLP.setLossRate((new
Integer(observed_loss_rate)).intValue());
          observed_utilization = rs3.getString("OBSERVED_THROUGHPUT");
          observed_throughput = (new
Integer(observed_utilization)).intValue()
            / 100 * (new Integer(target_throughput)).intValue() / 10;
          mySLP.setThroughput(observed_throughput);
        }
        slps_in_path.addElement(mySLP);
      }
      statement1.close();
      statement2.close();
      statement3.close();
    }catch(SQLException e){
      gui.sendText("PIB: getSLPsOfPath: SQLException: " + e);
    }
    if (showComments) {
      gui.sendText("PIB: getSLPsOfPath: SLP vector for path "
        + path_id + ":");
      gui.sendText(slps_in_path.toString());
    }
```

```
    return slps_in_path;
  }


  /**
   * Records the calculated effective quality of service parameters for
a
   * particular path.
   * @param path_id The id of the path in question.
   * @param effectiveDelay The effective delay that can be expected when
   * transmiting a flow over this path.
   * @param effectiveLossRate The effective loss rate that can be
expected when
   * transmiting a flow over this path.
   * @param effectiveThroughputRemaining The effective throughput
capacity that
   * was not being used at last observation.
   */
  public void setEffectiveQoSOfPath(int path_id, int effectiveDelay,
    int effectiveLossRate, int effectiveThroughputRemaining){
    Statement statement;
    ResultSet rs;
    try{
      statement = con.createStatement();
      statement.executeUpdate("UPDATE PATH SET EFFECTIVE_DELAY = '"
        + effectiveDelay + "', EFFECTIVE_LOSS_RATE ='" +
effectiveLossRate
        + "', EFFECTIVE_THROUGHPUT ='" + effectiveThroughputRemaining
        + "' WHERE PATH_ID = '" + path_id +"';");
      statement.close();
    }catch(SQLException e){
      gui.sendText("PIB: setEffectiveQoSOfPath: SQLException: " + e);
    }
    if (showComments) {
      gui.sendText("PIB: setEffectiveQoSOfPath: path "+path_id+"
effectively has "
        + " D = " + effectiveDelay + "ms, LR = " + effectiveLossRate
        + "%, remaining T = " + effectiveThroughputRemaining+"kbps");
    }
  }

  /**
   * Retrieves a vector of all path ids that travses the specified SLP.
   * @param address The IPv6 address of an interface.
   * @param service_level The level of service that this logical pipe is
   * providing.
   * @returns path_ids All of the path ids that traverse this SLP.
   */
  public Vector getAllPathIdsThatTraverseSLP(IPv6Address address,
                                                           int
service_level){
   Statement statement1,statement2;
   ResultSet rs1,rs2;
   String node_id = new String();
   Integer path_id;
   Vector path_ids = new Vector();
   try{
      statement1 = con.createStatement();
      statement2 = con.createStatement();
      rs1 = statement1.executeQuery("SELECT NODE_ID FROM INTERFACES
WHERE "
```

```
        + "ADDRESS = '" + address.toString() + "';");
      while (rs1.next()){
        node_id = rs1.getString("NODE_ID");
      }
      rs2 = statement2.executeQuery("SELECT PATH_ID FROM SLP_SEQUENCE
WHERE "
        + " LINK_ID = '" + address.getNetworkAddress() + "' AND SLP_ID =
'"
        + service_level + "' AND SOURCE_ROUTER = '" + node_id + "';");
      while (rs2.next()){
        path_id = new Integer(rs2.getString("PATH_ID"));
        path_ids.addElement(path_id);
      }
      statement1.close();
      statement2.close();
    }catch(SQLException e){
      gui.sendText("PIB: getAllPathIds: SQLException: " + e);
    }
    if (showComments) {
      gui.sendText("PIB: getAllPathIds: paths that traverse " + address
        + "'s service level #" + service_level);
      gui.sendText(path_ids.toString());
    }
    return path_ids;
  }

//**********************************************************************
*******
// These methods are used to retrieve various other info
//**********************************************************************
******/

  /**
   * Retrieves a vector of all interface addresses attached to this
router.
   * @param node_id The node id of the router in question.
   * @returns IPv6Addresses The interface addresses of this node.
   */
  public Vector getRouterInterfaces(int node_id){
    Statement statement;
    ResultSet rs;
    Vector IPv6Addresses = new Vector();
    IPv6Address address;
    try{
      statement = con.createStatement();
      //search for existing routers
      rs = statement.executeQuery("SELECT * FROM INTERFACES WHERE
NODE_ID = '"
        + node_id + "';");
      while (rs.next()) {
        address = IPv6Address.getByName(rs.getString("ADDRESS"));
        IPv6Addresses.addElement(address);
      }
      statement.close();
    }catch(SQLException e){
      gui.sendText("PIB: getRouterInterfaces: SQLException: " + e);
    }
    catch(UnknownHostException uhe){
      gui.sendText("PIB: getRouterInterfaces:UnknownHostException: "
        + uhe);
    }
```

```
    if (showComments) {
      gui.sendText("PIB: getRouterInterfaces: interface of node "
        +node_id+":");
      gui.sendText(IPv6Addresses.toString());
    }
    return IPv6Addresses;
  }

  /**
   * Deletes a specified router from the PIB.
   * @param node_id The node_id of the router to be deleted.
   */
  public void deleteARouter(int node_id){
    Statement statement1,statement2,statement3,statement4;
    ResultSet rs1,rs2;
    String link_id;
    try{
      statement1 = con.createStatement();
      statement2 = con.createStatement();
      statement3 = con.createStatement();
      statement4 = con.createStatement();
      rs1 = statement1.executeQuery("SELECT LINK_ID FROM INTERFACES
WHERE NODE_ID = '"
        + node_id + "';");
      while (rs1.next()){
        link_id = rs1.getString("LINK_ID");
        statement4.executeUpdate("DELETE FROM ROUTER WHERE NODE_ID = '"
          + node_id + "';");
        rs2 = statement2.executeQuery("SELECT NODE_ID FROM INTERFACES
WHERE LINK_ID = '"
          + link_id + "';");
        if (!rs2.next()){
          statement3.executeUpdate("DELETE FROM LINK WHERE LINK_ID = '"
            + link_id + "';");
        }
      }
      statement1.close();
      statement2.close();
      statement3.close();
      statement4.close();
    }catch(SQLException e){
      gui.sendText("PIB: deleteARouter: SQLException: " + e);
    }
    if (showComments) {
      gui.sendText("PIB: deleteARouter: deleting node " + node_id);
    }
  }

  /**
   * Retrieves a vector of all physical link ids known to the PIB.
   * @returns links All of the known links in the network.
   */
  public Vector getAllLinkIds(){
    Statement statement;
    ResultSet rs;
    IPv6Address link_id;
    Vector links = new Vector();
    try{
      statement = con.createStatement();
      rs = statement.executeQuery("SELECT * FROM LINK");
      while (rs.next()){
```

```
          link_id = IPv6Address.getByName(rs.getString("LINK_ID"));
          links.addElement(link_id);
        }
        statement.close();
      }catch(SQLException e){
        gui.sendText("PIB: getAllLinkIds: SQLException: " + e);
      }
      catch(UnknownHostException uhe) {
        gui.sendText("PIB: getAllLinkIds: " + uhe);
      }
      if (showComments) {
        gui.sendText("PIB: getAllLinkIds: known links addresses:");
        gui.sendText(links.toString());
      }
      return links;
    }

    /**
     * Retrieves a vector of all routers attached to a specific phyical
link.
     * @param link_id The IPv6 address of the link in question.
     * @returns routerIds The ids of routers that are directly attached to
this
     * link.
     */
    public Vector findRoutersOnLink(IPv6Address link_id){
      Statement statement;
      ResultSet rs;
      Vector routerIds = new Vector();
      int node_id;
      try{
        statement = con.createStatement();
        //search for existing routers
        rs = statement.executeQuery("SELECT * FROM INTERFACES WHERE
LINK_ID = '"
          + link_id.toString() + "';");

        // if any are found, then add them
        while (rs.next()) {
          node_id = (new Integer(rs.getString("NODE_ID"))).intValue();
          gui.sendText("PIB: findRoutersOnLink: node_id = " + node_id);
          routerIds.addElement(new Integer(node_id));
        }
        statement.close();
      }catch(SQLException e){
        gui.sendText("PIB: findRoutersOnLink: SQLException: " + e);
      }
      if (showComments) {
        gui.sendText("PIB: findRoutersOnLink: router on link "+link_id);
        gui.sendText(routerIds.toString());
      }
      return routerIds;
    }

}
```

```
package saam.server;

import saam.net.*;
import saam.message.*;
import java.sql.*;
import java.util.*;
import java.io.*;

/**
 * The <em>PathInformationBase</em> is an abstract class within the SAAM
 * architecture that dictates what methods need to be implemented by any
class
 * that wishes to perform the retrieval and storage activities involved
with
 * maintaining a picture of the network.
 */
public abstract class PathInformationBase {

  /**
   * Removes all current path data from the database.. Its most commonly
used
   * during initialization of a SAAM server for a new network.
   */
  public abstract void deleteAllData();

//*********************************************************************
*******
// These methods are used to process link state advertisements from
routers
//*********************************************************************
******/

  /**
   * Determines whether a given router exists yet within the PIB.
   * @param IPv6Addresses A vector of interface addresses contained
within
   * a Hello or an LSA message.
   * @returns node_id The id of the node containing at least one of the
   * interface addresses in the vector that was passed.
   */
  public abstract int doesRouterExist(Vector IPv6Addresses);

  /**
   * Finds an unassigned node id and adds it to the PIB. It is commonly
used
   * for assigning a new node_id to a previously unknown router.
   * @returns max_node_id An unassigned router id.
   */
  public abstract int getNewNodeId();

  /**
   * Determines whether a given interface exists yet within the PIB.
   * @param myIPv6Address The interface address contained within a hello
or LSA
   * message.
   * @returns found True if the interface address already exists within
the PIB.
   */
  public abstract boolean doesInterfaceExist(IPv6Address myIPv6Address);

  /**
```

```
    * Determines whether a given link exists yet within the PIB.
    * @param address The IPv6 address of an interface.
    * @returns found True if the link address already exists within the
PIB.
    */
  public abstract boolean doesLinkExist(IPv6Address address);

  /**
    * Adds a new link to the PIB.
    * @param address The IPv6 address of an interface.
    * @param max_bandwidth The max transmission rate over this network
segment.
    */
  public abstract void addLink(IPv6Address address, int max_bandwidth);

  /**
    * Adds a new interface to the PIB.
    * @param node_id The id of the router whose interface is being added.
    * @param address The IPv6 address of an interface.
    */
  public abstract void addInterface(int node_id, IPv6Address address);

  /**
    * Determines whether a service level pipe exists yet within the PIB.
    * @param address The IPv6 address of an interface.
    * @param service_level The level of service that this logical pipe is
    * providing.
    * @returns found True if this SLP is already in the PIB.
    */
  public abstract boolean doesSLPExist(IPv6Address address, int
service_level);

  /**
    * Updates the status of a known SLP's delay, loss_rate, and
throughput.
    * @param address The IPv6 address of an interface.
    * @param service_level The level of service that this logical pipe is
    * providing.
    * @param delay The average delay experienced by a packet's stay in
the
    * particular SLP outbound queue.
    * @param loss_rate The average loss_rate experienced by packets in a
    * particular SLP.
    * @param throughput The average throughput provided by a particular
SLP.
    */
 public abstract void updateSLP(IPv6Address address, int service_level,
   int delay, int loss_rate, int throughput);

  /**
    * Adds a previously unknown SLP to the PIB along with its targeted
QoS.
    * @param address The IPv6 address of an interface.
    * @param service_level The level of service that this logical pipe is
    * providing.
    * @param target_delay The average delay experienced by a packet's
stay in the
    * particular SLP outbound queue.
    * @param target_loss_rate The average loss_rate experienced by
packets in a
    * particular SLP.
```

```
   * @param target_throughput The average throughput provided by a
particular SLP.
   */
  public abstract void addSLP(IPv6Address address, int service_level,
int target_delay,
     int target_loss_rate, int target_throughput);

//***********************************************************************
*******
// These methods are used to process a flow request from a host
//***********************************************************************
******/

  /**
   * Finds a router id that has an interface to on the same link as the
host
   * making a flow request.
   * @param address The IPv6 address of the interface of the host
requesting
   * the flow.
   * @returns ARouter The router id of the first router found on this
link.
   */
  public abstract int findARouterOnLink(IPv6Address address);

  /**
   * Determines if there is a path that can support a particular flow
request.
   * A value of zero is returned if no path can support this QoS.
   * @param source_router The node id of a router on the same physical
link as
   * the source host.
   * @param destination_router The node id of a router on the same
physical
   * link as the destination host.
   * @param myFlowRequest A host's request for the establishment of a
flow.
   * @returns path_id The id of a path that can support this request.
   */
  public abstract int getPathThatCanSupportFlowRequest(int
source_router,
     int destination_router, FlowRequest myFlowRequest);

  /**
   * Finds an unassigned flow id and assigns this new flow to a path.
   * @param path_id The id of a path that can support this request.
   * @param source_router The node id of a router on the same physical
link as
   * the source host.
   * @param destination_router The node id of a router on the same
physical
   * link as the destination host.
   * @param myFlowRequest A host's request for the establishment of a
flow.
   * @returns max_flow_id The id that is being assigned to this flow.
   */
  public abstract int getNewFlowId(int path_id, int source_router,
                          int destination_router, FlowRequest
myFlowRequest);

  /**
```

```
   * Retrieves the sequence of SLPs that make up a given path.
   * @param path_id The id of the path in question.
   * @returns slps_in_path A vector of SLPs that compose this path.
   */
  public abstract Vector getSLPSequenceOfPath(int path_id);

  /**
   * Retrieves  the IPv6 address of an interface.
   * @param node_id The id of the router whose interface is being
queried.
   * @param link_id The network portion of the IPv6 address of an
interface.
   * @returns address The address of the interface that connects this
node and
   * link.
   */
  public abstract IPv6Address getInterfaceAddress(int node_id,
                                                          IPv6Address
link_id);

//*************************************************************************
*******
// These methods are used to determine all possible paths across the
network
//*************************************************************************
******/

  /**
   * Retrieve all of the router ids assigned by the PIB so far.
   * @returns V A vector of all assigned router ids.
   */
  public abstract Vector getAllRouterIds();

  /**
   * Retrieves the maximum service level of this SAAM region.
   * @returns max_slp_id The numerically highest service level id
assigned.
   */
  public abstract int findMaxServiceLevel();

  /**
   * Retrieves an array of parents for each router.  A parent is a
directly
   * connected node.
   * @param V A vector of all router ids.
   * @param service_level The level of service for which paths are being
built
   * for.
   * @returns parent A hashtable of vectors containing the parents of
each router.
   */
  public abstract Hashtable getParents(Vector V, int service_level);

  /**
   * Assigns a path from a source and to a destination.
   * @param source_router The node_id of the source of the path.
   * @param destination_router The node_id of the destination of the
path.
   * @returns max_path_id The id to be assigned to this new path.
   */
```

```
  public abstract int getNewPathId(int source_router, int
destination_router);

  /**
   * Identifies the id of the physical link between two adjacent
routers. If no
   * link exists between a source and destination router, the default
address
   * of all zeros is returned.
   * @param source_router The node_id of a router.
   * @param destination_router The node_id of an adjacent router.
   * @returns subnet The link id between this source and destination.
   */
  public abstract IPv6Address getLinkBetween(int source_router,
                                                         int
destination_router);

  /**
   * Assigns a service level pipe sequence entry in the building of a
path.
   * @param service_level The level of service for which paths are being
built
   * for.
   * @param source_router The node_id of the source of the SLP.
   * @param link_id The subnet that this SLP goes over.
   * @param path_id The id assigned to the path.
   * @param sequence_number The number assigned to specify the sequence
of this
   * SLP in the path.
   */
  public abstract void assignSLPSequence(int service_level,
    int source_router, IPv6Address link_id,
    int path_id, int sequence_number);

//**********************************************************************
*******
// These methods are used to determine the effective QoS of paths
//**********************************************************************
******/

  /**
   * Retrieves a vector of all path ids constructed by the PIB.
   * @returns path_ids All of the path ids known to the PIB.
   */
  public abstract Vector getAllPathIds();

  /**
   * Retrieves the SLPs that make up a given path.
   * @param path_id The id of the path in question.
   * @returns slps_in_path The SLPs that make up the path.
   */
  public abstract Vector getSLPsOfPath(int path_id);

  /**
   * Records the calculated effective quality of service parameters for
a
   * particular path.
   * @param path_id The id of the path in question.
   * @param effectiveDelay The effective delay that can be expected when
   * transmiting a flow over this path.
```

```
   * @param effectiveLossRate The effective loss rate that can be
expected when
   * transmiting a flow over this path.
   * @param effectiveThroughputRemaining The effective throughput
capacity that
   * was not being used at last observation.
   */
  public abstract void setEffectiveQoSOfPath(int path_id, int
effectiveDelay,
    int effectiveLossRate, int effectiveThroughputRemaining);

  /**
   * Retrieves a vector of all path ids that travses the specified SLP.
   * @param address The IPv6 address of an interface.
   * @param service_level The level of service that this logical pipe is
   * providing.
   * @returns path_ids All of the path ids that traverse this SLP.
   */
public abstract Vector getAllPathIdsThatTraverseSLP(IPv6Address address,
                                                              int
service_level);


//*********************************************************************
*******
// These methods are used to retrieve various other info
//*********************************************************************
******/

  /**
   * Retrieves a vector of all interface addresses attached to this
router.
   * @param node_id The node id of the router in question.
   * @returns IPv6Addresses The interface addresses of this node.
   */
  public abstract Vector getRouterInterfaces(int node_id);

  /**
   * Deletes a specified router from the PIB.
   * @param node_id The node_id of the router to be deleted.
   */
  public abstract void deleteARouter(int node_id);

  /**
   * Retrieves a vector of all physical link ids known to the PIB.
   * @returns links All of the known links in the network.
   */
  public abstract Vector getAllLinkIds();

  /**
   * Retrieves a vector of all routers attached to a specific phyical
link.
   * @param link_id The IPv6 address of the link in question.
   * @returns routerIds The ids of routers that are directly attached to
this
   * link.
   */
  public abstract Vector findRoutersOnLink(IPv6Address link_id);

}
```

```
package saam.server;

import saam.net.*;
import saam.message.*;
import saam.control.*;
import saam.router.*;
import saam.util.*;
import java.net.*;
import java.util.*;
import java.io.*;

/**
 * The <em>Server</em> is an object within the SAAM architecture that
 * maintains a picture of the network for use in assigning flows to
paths.
 */

public class Server {

  //declare class variables

  /** Contains what is known about the network. */
  private PathInformationBase PIB;

  /** Enables the Server to receive and send particular types of
messages. */
  private ControlExecutive controlExec;

  /** A maximum number of hops that a search for different paths may
take. */
  private int Hmax = 4;

  /**
   * Used to lookup what flow id should be used to send out control
messages
   * to specified routers.
   */
  private Hashtable flowLookUp = new Hashtable();

  /** Used to assign the right number of service level pipes to
interfaces in
   * this SAAM region. Only used during initialization -- later were
assume
   * SLPs are known to routers
   */
  private int numOfServiceLevels = 4;

  /**
   * The value assigned to flow ids that can not be supported. This
should be
   * switched over to 0 as soon as routers are converted.
   */
  public static int FLOWNOTSUPPORTABLE = 99;

  public static int INITIALDELAY = 0;
  public static int INITIALLOSSRATE = 0;
  public static int INITIALTHROUGHPUT = 10000;

  public static int RETURNFLOWDELAY = 50;
  public static int RETURNFLOWLOSSRATE = 50;
  public static int RETURNFLOWTHROUGHPUT = 1000;
```

```
  public static int ROUTERNOTINPIB = 0;

  public static int NOSUPPORTABLEPATHINPIB = 0;

  public static int SERVERNODEID = 1;

  public static int FLOWTOSERVER = 0;

  public static int PSUEDORANDOMSOURCEPORT = 8000;

  public static int INITIALPATHID = 0;

  public static int INITIALHEIGHTOFSEARCH = 1;
  public static int INCREMENTATIONOFSEARCH = 1;
  public static int DESTINATIONNODE = 0;

  public static int INITIALZERO = 0;


  /** Defines with the appropriate IPv6 address of this server. */
  //private String serverIPv6 = controlExec.getServerIP().toString();
  private String serverIPv6 = "99.99.99.0.0.0.0.0.0.0.0.0.0.0.0.1";

  /** Time when the all possible paths were found. */
  private long timeOfLastPIBBuild = System.currentTimeMillis();

  /**
   * The amount of time that we want to have between rebuilding of
paths. This
   * is not currently implemented.
   */
  private long timeBetweenPIBBuilds = 120000; // 2 minutes (or 120 sec)

  /** A boolean that will allow the showing of comments. */
  private boolean showComments = true;

  private SAAMRouterGui gui;

  /**
   * Constructs a server that will use a specified type of <em>Path
Information
   * Base</em>.  The PIB may be in the form of a database structure
(which
   * requires an existing ODBC configured local database) or a class
   * object structure. The control executive is the interface to the
IPv6
   * protocol stack, in order for messages to flow to and from the
network.
   * The final step taken is the deletion of all existing data, which is
   * important only in a database structure since a class object
structure is
   * volatile.
   * @param type The type of structure that the PIB is to assume.
   * @param controlExec The control executive that will exchange
messages
   * with this server.
   */
  public Server(String type, ControlExecutive controlExec){
    if (type == "database")
      PIB = new DatabaseStructure();
```

```
    else
      PIB = new ClassObjectStructure();

    this.controlExec = controlExec;
    gui=new SAAMRouterGui("Server");

    PIB.deleteAllData();
  }

//*********************************************************************
*******
// These methods handle external network communications from routers
//*********************************************************************
******/


  /**
   * Receives Hello messages from routers and then processes them.  It
starts
   * building a vector of IPv6Addresses from the interfaces included in
the
   * Hello message. This vector is passed to the PIB's doesRouterExist()
which
   * determines if a router with any of these interfaces have been
identified
   * before. If this is a new router, a new unique node id is
assigned.<p>
   * For each of the interfaces identified in the Hello message, if this
   * interface was is not known to the PIB, check to see if the
corresponding
   * link is known to the PIB. If this link is not known to the PIB, add
it.
   * Next, add the new interface between the node and link. Also, add
each
   * service level pipe that is assigned within this SAAM region.<p>
   * The next step is to rebuild the paths that are possible across the
network
   * now considering this new hello message. The frequency of these
rebuilds is
   * not a major concern in a controlled environment, but will need to
be
   * addressed later. Finally, a flow request is create and received for
   * communicating back to this node. This is only possible if the PIB's
   * determineAllPossiblePaths() has been executed after the processing
of this
   * particular hello message, if this a new router. After all paths to
each
   * known router are found, we finish this method with a call to
   * determineEffectiveQoSForPaths(). The call to
   * determineEffectiveQoSForPaths() ensures that even if no QoS
parameters are
   * known about these new parts of the network, that at least some
initial
   * values will be assigned. This initialization allows the new paths
to be
   * assigned if needed.
   * @param hello An initialization message from a router.
   */
  public void processHello(Hello hello) {

    long start, finish;
```

```
    Vector interfaces;
    int node_id = INITIALZERO;
    InterfaceID myInterface;
    int bandwidth = INITIALZERO;
    IPv6Address address = new IPv6Address();
    Vector IPv6Addresses = new Vector();
    boolean newRouter = true;
    FlowRequest myFlowRequest = new FlowRequest();

    // capture the start time of processing a hello
    start = System.currentTimeMillis();

    // produce a vector of IPv6Addresses
    interfaces = hello.getInterfaceIDs();
    for (int i = INITIALZERO; i < interfaces.size(); i++){
      address = ((InterfaceID)interfaces.elementAt(i)).getIPv6();
      IPv6Addresses.addElement(address);
    }

    // check if router exists and if so, return it's node id, else
return 0
    node_id = PIB.doesRouterExist(IPv6Addresses);

    // if the router does not exist in PIB
    if (node_id == ROUTERNOTINPIB){
      // assign it a new node id
      node_id = PIB.getNewNodeId();
    } else {
      newRouter = false;
    }

    // run through all of the LSA interfaces
    for (int i = INITIALZERO; i < interfaces.size(); i++) {
      myInterface = (InterfaceID)interfaces.elementAt(i);
      address = myInterface.getIPv6();
      // if a new interface is not found in the PIB, then ...
      if (!PIB.doesInterfaceExist(address)){
        bandwidth = myInterface.getBandwidth();
        address = myInterface.getIPv6();
        // if the link is not contained in the PIB, then add it
        if (!PIB.doesLinkExist(address)){
          PIB.addLink(address, bandwidth);
        }
        // now add the interface between the node and the link
        PIB.addInterface(node_id, address);
        // now add each service level pipe
        for (int service_level = 0; service_level < numOfServiceLevels;

service_level++){
          PIB.addSLP(address, service_level, INITIALDELAY,
INITIALLOSSRATE,
            INITIALTHROUGHPUT);
        }
      } // end if
    } //end interfaces for

    // capture the hello processing finish time
    finish = System.currentTimeMillis();
      gui.sendText("Server: processHello: Time required = "
        +(finish-start)+" milliseconds.");
```

```
    // time since last PIB build is > 2 min and if node did not exist
before
    //if ((timeOfLastPIBBuild - System.currentTimeMillis())
>timeBetweenPIBBuilds
    //                                                                     &&
newRouter){

      // rebuild all possible paths
      findAllPossiblePaths();

      // determine effective QoS of each path
      determineEffectiveQoSForPaths();

      // construct a new flow to this router
      try{
        myFlowRequest = new
FlowRequest(IPv6Address.getByName(serverIPv6),
          address,System.currentTimeMillis(),RETURNFLOWDELAY,
          RETURNFLOWLOSSRATE,RETURNFLOWTHROUGHPUT);
      } catch(UnknownHostException uhe){
        System.err.println("Server: main: UnknownHostException: " +
uhe);
      }
      processFlowRequest(myFlowRequest);
    //}

  } //end processFlowRequest


  /**
   * Receives link state advertisement messages from router and
processes the
   * service level pipe status information that they contain. It begins
by
   * checking to see if a router with the interface address described by
this
   * LSA is known to the PIB. If such a router is known to exist, it
then
   * checks to see if the service level pipe described by this LSA is
known to
   * the PIB. If the service level pipe is known, then update its
status.
   * Otherwise, add the SLP with the specified QoS characteristics.
Finally,
   * update the effective QoS for the paths that pass over this service
level
   * pipe by calling the determineEffectiveQoSForPaths().
   * @param router A representation of a router as defined by an LSA.
   */
  public void processLSA(LinkStateAdvertisement LSA) {

    long start, finish;
    int node_id = INITIALZERO;
    int bandwidth = INITIALZERO;
    byte service_level = 0;
    int delay = INITIALZERO;
    int loss_rate = INITIALZERO;
    int utilization = INITIALZERO;
    Vector interfaces = new Vector(3,1);
    Vector SLPs = new Vector(3,1);
    IPv6Address link_id;
```

```
    IPv6Address address;
    Vector IPv6Addresses = new Vector(1,1);

    // capture the start time of processing an LSA
    start = System.currentTimeMillis();

    // produce a one element vector of IPv6Addresses
    address = LSA.getMyIPv6();
    IPv6Addresses.addElement(address);

    // check if router exists and if so, return it's node id, else
return 0
    node_id = PIB.doesRouterExist(IPv6Addresses);

    // if the router does exist in PIB, then so does the interface...
    if (node_id != ROUTERNOTINPIB){

      service_level = LSA.getServiceLevel();
      delay = LSA.getDelay();
      loss_rate = LSA.getLossRate();
      utilization = LSA.getUtilization();

      if (showComments){
        gui.sendText("Server: processLSA: node_id = " + node_id
          + ", address = " + address + ", SL = "+service_level
          +", D = " +delay+ ", LR = "+loss_rate
          +", U = "+utilization);
      }

      // if this SLP is defined, then just update its status
      if(PIB.doesSLPExist(address, service_level))
      {
        PIB.updateSLP(address, service_level, delay, loss_rate,
utilization);
      }
      // otherwise, insert it
      else {
        PIB.addSLP(address, service_level, delay, loss_rate,
utilization);
      } // end else
    } // end if
    else { //do nothing
    }

    // capture the LSA processing finish time
    finish = System.currentTimeMillis();
    gui.sendText("Server: processLSA: Time required = "
      +(finish-start)+" milliseconds.");

    // revise the effective QoS of paths made up of this SLP
    determineEffectiveQoSForPaths(address,service_level);

  } //end processLSA


  /**
   * Receives and processes flow requests from applications. It begins
   * by finding a source and a destination router. These routers may be
where
   * the applications are residing themselves, which is our standard
situation.
```

```
    * The application could, however, reside on some host that is not
registered
    * with the PIB as a router. In this case, the appropriate source or
    * destination router would be a router connected to the same link.
<p>
    * The PIB is checked to ensure that there is the effective QoS
available on
    * some path to satisfy the request. If a satisfactory path is found,
a new
    * unique flow id is assigned and this new flow is associated with
that path.
    * Each router in the path is retrieved and a new flow routing table
entry is
    * sent to each. If no path can provide the requested level of QoS,
then the
    * flow is assigned to zero, which will be interpreted by IPv6 as best
effort
    * traffic. Finally, a flow response is sent back to the application
to
    * inform it of its assigned flow id. If the flow id that is return is
zero,
    * it will be the application's responsibility to either lower it QoS
request
    * or to send its traffic as best effort.
    * @param flow_request The message requesting the establishment of a
flow.
    */
   public void processFlowRequest(FlowRequest flow_request) {

     /** A vector of slp_sequence information for a path. */
     Vector slps_in_path;
     SLPSequence currentSLPSequence,nextSLPSequence = new SLPSequence();
     int SLP_source_router, SLP_destination_router, service_level;
     IPv6Address link_id = new IPv6Address();
     IPv6Address next_hop;
     IPv6Address sourceAddress;
     int source_router, destination_router, path_id,
       flow_id=FLOWNOTSUPPORTABLE;
     long start, finish;

     // capture the start time of processing a flow request
     start = System.currentTimeMillis();

     // find a router on the same subnet as the source host
     source_router =
       PIB.findARouterOnLink(flow_request.getSourceInterface());

     // find a router on the same subnet as the destination host
     destination_router =
       (PIB.findARouterOnLink(flow_request.getDestinationInterface()));

     path_id = PIB.getPathThatCanSupportFlowRequest(source_router,
                                                    destination_router,
flow_request);

     // if a path can support this request, then...
     if(path_id != NOSUPPORTABLEPATHINPIB){

       // assign a flow id to the request
       flow_id =
PIB.getNewFlowId(path_id,source_router,destination_router,
```

```
flow_request);

      // determine each router in path
      // transmit Flow Routing Table Entry to it
      slps_in_path = PIB.getSLPSequenceOfPath(path_id);
      // for each router in the path, send a FRTE update
      for (int index = INITIALZERO; index < slps_in_path.size();
index++){
        // assign new slp sequence object
        currentSLPSequence = (SLPSequence)slps_in_path.elementAt(index);

        // if not the last link..
        if (index+1 != slps_in_path.size()){
          nextSLPSequence =
(SLPSequence)slps_in_path.elementAt(index+1);
        }

        // retrieve values from this object
        SLP_source_router = currentSLPSequence.getSourceRouter();
        link_id = currentSLPSequence.getLinkId();
        service_level = currentSLPSequence.getServiceLevel();

        // if not the last link...
        if (index+1 != slps_in_path.size()){
          SLP_destination_router = nextSLPSequence.getSourceRouter();
        } else {
          // else it is the destination node of the flow
          SLP_destination_router = destination_router;
        }
        // determine destination address for next hop
        next_hop = PIB.getInterfaceAddress(SLP_destination_router,
link_id);

        // determine source address
        sourceAddress = PIB.getInterfaceAddress(SLP_source_router,
link_id);

        // send the flow routing table entry update
        sendFRTEUpdate(sourceAddress, flow_id, next_hop, service_level);

      } // end for

    } // end if

    //give routers time to finish updating tables
    try{
      Thread.sleep(2000);
    }catch(InterruptedException ie){
      gui.sendText(ie.toString());
    }
     // if the source of this flow is the server,
    if (source_router == SERVERNODEID) {
      // then add this new flow to hash table for later lookup
      if (showComments){
        gui.sendText("Server: processFlowRequest: use flow "+flow_id
          +" to send to node "+destination_router);
      }
      if (destination_router == SERVERNODEID) {
        flow_id = FLOWTOSERVER;
      }
```

```
      flowLookUp.put(new Integer(destination_router),new
Integer(flow_id));
    }

    sendFlowResponse(flow_request, flow_id);

    // capture the flow request processing finish time
    finish = System.currentTimeMillis();
    gui.sendText("Server: processFlowRequest: Time required = "
      +(finish-start)+" milliseconds.");
  }

  /**
   * Receives flow termination from routers and then processes them.
   */
  public void receiveFlowTermination() { }


//*********************************************************************
*******
// These methods handle external network communications to routers
//*********************************************************************
******/

  /**
   * Sends a flow routing table entry update message to a router. This
message
   * provides the router the required information to forward packets
based on
   * its flow id.
   * @param sourceAddress The router that will receive the FRTE update.
   * @param flow_id The id assigned to the flow in question.
   * @param next_hop The IPv6 address of the next node in the path.
   * @param service_level The service level that this flow is assigned
to.
   */
  public void sendFRTEUpdate(IPv6Address sourceAddress, int flow_id,
                                    IPv6Address next_hop, int
service_level) {
    if(showComments){
      gui.sendText("Server: sendFRTEUpdate: flowLookUp hashtable:");
      gui.sendText(""+flowLookUp);
    }
    FlowRoutingTableEntry myFRTE = new FlowRoutingTableEntry(flow_id,
                                              (byte)service_level,
next_hop);
    int sourcePort = PSUEDORANDOMSOURCEPORT;

//controlExec.listenToRandomPort(this);
    short destPort = ControlExecutive.SAAM_CONTROL_PORT;
    IPv6Address destHost = sourceAddress;
    // take steps to determine what flow id to send the packet on
    Vector interfaces = new Vector();
    interfaces.addElement(destHost);
    int destNodeId = PIB.doesRouterExist(interfaces);
    int flowIdToSendItOn = ((Integer)flowLookUp.get
                                          (new
Integer(destNodeId))).intValue();
    try{
      controlExec.send(this,myFRTE, flowIdToSendItOn, (short)sourcePort,
```

```
                                                                 destHost,
destPort);
    } catch (FlowException fe){
      System.err.println(fe.toString());
    }
    if (showComments){
      gui.sendText("Server: sendFRTEUpdate: FRTE for flow " + flow_id
        + " sent to interface "+sourceAddress);
      gui.sendText("          with next hop= "+next_hop
        +" on service level "+service_level+" via flow
"+flowIdToSendItOn);
    }
  }

  /**
   * Sends a flow response to the requesting application to notify it of
   * its newly assigned flow id. A flow id of zero is used to indicate
that the
   * flow cannot be supported. Once a flow response message is
instantiated and
   * a source and destination port is defined, the control executive's
send()
   * is called to send it to the destination host.
   * @param flow_request The flow request message that was received.
   * @param flow_id The flow id that is assigned to the flow request.
   */
  public void sendFlowResponse(FlowRequest flow_request, int flow_id){
    if(showComments){
      gui.sendText("Server: sendFlowResponse: flowLookUp hashtable:");
      gui.sendText(""+flowLookUp);
    }
    FlowResponse response = new
FlowResponse(flow_request.getTimeStamp(),
      flow_id);
    int sourcePort = PSUEDORANDOMSOURCEPORT;

//controlExec.listenToRandomPort(this);
    short destPort = ControlExecutive.SAAM_CONTROL_PORT;
    IPv6Address destHost = flow_request.getSourceInterface();
    // take steps to determine what flow id to send the packet on
    Vector interfaces = new Vector();
    interfaces.addElement(destHost);
    int destNodeId = PIB.doesRouterExist(interfaces);
    int flowIdToSendItOn = ((Integer)flowLookUp.get
                                         (new
Integer(destNodeId))).intValue();
    try{
      controlExec.send(this, response, flowIdToSendItOn,
(short)sourcePort,
                                                                 destHost,
destPort);
    }catch(FlowException fe){
      System.err.println(fe.toString());
    }
    if (showComments){
      gui.sendText("Server: sendFlowResponse: Flow response "
        + response + " from SourcePort: "+sourcePort+" to "+destHost
        + " sent via flow "+flowIdToSendItOn);
    }

  }
```

```
//**********************************************************************
*******
// These methods handle internal manipulation of data describing network
status
//**********************************************************************
******/

  /**
   * Determines all of the possible paths that exist between any source
and
   * destination router in the network. This determination is based on
the
   * physical definition of the network that is provided by the hello
messages
   * received from the routers and stored within the PIB. The paths that
are
   * found are then recorded in the PIB for fast assignment of flows
later.<p>
   * All node ids are first retrieved from the PIB. For each service
level, we
   * build an array of parents of each node. A parent is node that is
directly
   * connected. Those directly connected nodes would have service level
pipes
   * that would need to be passed through to get to the child node in
question.
   * This parent array is used to populate a path table. Each node id is
   * assigned as the final destination of path and all of the different
paths
   * are then found by working out from this destination. For each of
these
   * destination nodes, a call is made to processPath() to find all the
valid
   * paths that go to this destination node. We make the call with a
specified
   * height of search of 1.
   */
  public void findAllPossiblePaths() {
    long start, finish;
    int NumberOfRouters;
    int max_slp_id = INITIALZERO;
    /** A count of the highest path id assigned so far. */
    int max_path_id = INITIALZERO;
    int service_level = INITIALZERO;

    /** A vector of the routers that are known by the db. */
    Vector V = new Vector();

    /** A vector of the parent routers for each given destination
router. */
    Hashtable parent;

    // capture the start time of processing a path data
    start = System.currentTimeMillis();

    // reset the maximum path id assigned so far to zero
    max_path_id = INITIALPATHID;

    V = PIB.getAllRouterIds();
```

```
    //retrieve COUNT of routers
    NumberOfRouters = V.size();

    //find all possible paths for each service level
    max_slp_id = (new Integer(PIB.findMaxServiceLevel())).intValue();
    for (service_level = INITIALZERO; service_level <= max_slp_id;
service_level++){

      //build parent array of each SLP at this service level
      parent = PIB.getParents(V, service_level);

      //populate path table
      for (int index = INITIALZERO; index < NumberOfRouters; index++){
        int heightOfSearch = INITIALHEIGHTOFSEARCH;
        int aPath[] = new int[Hmax + INCREMENTATIONOFSEARCH];
        aPath[DESTINATIONNODE] =
((Integer)V.elementAt(index)).intValue();
        processPath(parent, aPath, heightOfSearch,service_level);
      }
    }

    // capture the path data processing finish time
    finish = System.currentTimeMillis();
    gui.sendText("Server: findAllPossiblePaths: Time required = "
      +(finish-start)+" milliseconds.");

    timeOfLastPIBBuild = finish;

  }

  /**
   * Processes all valid paths that arrive at the destination node
within some
   * range of hops. For each parent of the node at the distance of
   * heightOfSearch from the destination, a check is made to ensure that
adding
   * this new parent will cause no cycle. If this checks out, then that
parent
   * can be added and a new path can be assigned. The service level
pipes in
   * this new path are identified and their sequence numbers in this
path are
   * recorded to the PIB. Next, a check is made to see if the height of
the
   * search is less than the server's max search height of Hmax. If it
is less,
   * the method recursively calls itself with an incremented
heightOfSearch
   * variable.
   * @param parent Contains each router and a list of other
   * routers that are directly attached to them.
   * @param aPath[] An array contain a path from a source node,
   * aPath[heightOfSearch], to a destination node, aPath[0].
   * @param heightOfSearch The number of nodes in the path so far.
   * @param service_level The level of service assigned to a flow.
   */

  public void processPath(Hashtable parent,
                          int aPath[], int heightOfSearch, int
service_level){
    IPv6Address link_id;
```

```
    int justARouter;
    int sequence_number;
    int path_id;
    Enumeration W = ((Vector)parent.get(
                                new Integer(aPath[heightOfSearch-
1]))).elements();
    while (W.hasMoreElements()) {
      justARouter = ((Integer)W.nextElement()).intValue();
      if (causeNoCycle(aPath, heightOfSearch, justARouter)) {

        // assign this router as the source in this path
        aPath[heightOfSearch] = justARouter;

        // record the new path id, etc.
        path_id = PIB.getNewPathId(justARouter, aPath[DESTINATIONNODE]);

        // run through the SLP's and record their sequence
        for (int index = heightOfSearch; index >DESTINATIONNODE; index--
){

          // determine link_id of this SLP
          link_id = PIB.getLinkBetween(aPath[index],
                                          aPath[index-
INCREMENTATIONOFSEARCH]);

          // assign the SLP its sequence number
          sequence_number = heightOfSearch - index;
          PIB.assignSLPSequence(service_level, aPath[index],
            link_id, path_id, sequence_number);
        }
        if (heightOfSearch < Hmax) {
          processPath(parent, aPath,
heightOfSearch+INCREMENTATIONOFSEARCH,

service_level);
        }
      }
    }
    if (showComments){
      gui.sendText("Server: processPath: paths at depth of
"+heightOfSearch
        +" from node "+aPath[DESTINATIONNODE]+" is completed.");
    }
  }

  /**
   * Checks to ensure that the addition of a specified new node to a
specified
   * path does not result in a cycle being created. This check is
completed by
   * the new node is already a member of the list of nodes in the path
already.
   * @param aPath[] An array contain a path from a source node,
   * aPath[heightOfSearch], to a destination node, aPath[0].
   * @param heightOfSearch The number of nodes in the path so far.
   * @param justARouter The proposed next node in for a new path.
   * @returns noCycles True if no cycles are created by the addition of
   * justARouter.
   */
  public boolean causeNoCycle(int aPath[], int heightOfSearch,
                              int justARouter){
```

```
      boolean noCycles = true;
      for (int index = INITIALZERO; index < heightOfSearch; index++){
        if (justARouter == aPath[index]){
          if (showComments){
            gui.sendText("Server: causeNoCycle: adding "+justARouter
              +" to get to "+aPath[DESTINATIONNODE]+" via "
              +aPath[heightOfSearch-INCREMENTATIONOFSEARCH]
              +" at a height of "+heightOfSearch+" caused cycle!");
          }
          return noCycles = false;
        }
      }
      if (showComments){
        gui.sendText("Server: causeNoCycle: adding "+justARouter
          +" as hop #"+heightOfSearch+" to get to "+aPath[DESTINATIONNODE]
          +" via "+aPath[heightOfSearch-INCREMENTATIONOFSEARCH]
          +" does not cause cycle.");
      }
      return noCycles;
    }

    /**
     * Determines what the effective QoS on each path in the PIB is. For
each
     * path, the service level pipes that compose it are retrieved. Then,
for
     * each of these service level pipes, we total up the delay and loss
rate.
     * The effective throughput remaining is determined by finding the
minimum
     * difference between the observed throughput and the target
throughput of
     * each service level pipe.
     */
    public void determineEffectiveQoSForPaths(){
      long start, finish;
      Vector path_ids;
      Integer myPathId;
      Vector SLPs;
      SLP mySLP;
      int totalDelay = INITIALZERO, totalLossRate = INITIALZERO,
        throughput = INITIALZERO, targetThroughput = INITIALZERO,
        throughputRemaining = INITIALZERO,
        minThroughputRemaining = INITIALZERO;

      // capture the start time of processing a path data
      start = System.currentTimeMillis();

      // for each path
      path_ids = PIB.getAllPathIds();

      for (int index1 = INITIALZERO; index1 < path_ids.size(); index1++){

        // for each path
        myPathId = (Integer)path_ids.elementAt(index1);

        SLPs = PIB.getSLPsOfPath(myPathId.intValue());

        for (int index2 = INITIALZERO; index2 < SLPs.size(); index2++){

          mySLP = (SLP)SLPs.elementAt(index2);
```

```
        // add delay to total delay
        totalDelay = totalDelay + mySLP.getDelay();

        // add loss rate to total loss rate
        totalLossRate = totalLossRate + mySLP.getLossRate();

        // find min throughput
        throughput = mySLP.getThroughput();

        targetThroughput = mySLP.getTargetThroughput();

        throughputRemaining = targetThroughput - throughput;
        if (throughputRemaining < minThroughputRemaining ||
                                 minThroughputRemaining == INITIALZERO){
          minThroughputRemaining = throughputRemaining;
        }

      }


PIB.setEffectiveQoSOfPath(myPathId.intValue(),totalDelay,totalLossRate,

minThroughputRemaining);
      totalDelay = INITIALZERO;
      totalLossRate = INITIALZERO;
      minThroughputRemaining = INITIALZERO;
    }

    // capture the path data processing finish time
    finish = System.currentTimeMillis();
    gui.sendText("Server: determineEffectiveQoSForPaths: Time required =
"
      +(finish-start)+" milliseconds.");

  }

  /**
   * Determines the effective QoS for just those paths that pass over
the
   * specified service level pipe. For each path, the service level
pipes that
   * compose it are retrieved. Then, for each of these service level
pipes, we
   * total up the delay and loss rate. The effective throughput
remaining is
   * determined by finding the minimum difference between the observed
   * throughput and the target throughput of each service level pipe.
   * @param address The address of the interface containing this service
level.
   * @param service_level The service level of this SLP.
   */
  public void determineEffectiveQoSForPaths(IPv6Address address, int
service_level){
    long start, finish;
    Vector path_ids;
    Integer myPathId;
    Vector SLPs;
    SLP mySLP;
    int totalDelay = INITIALZERO, totalLossRate = INITIALZERO,
```

```
      throughput = INITIALZERO, targetThroughput = INITIALZERO,
      throughputRemaining = INITIALZERO, minThroughputRemaining =
INITIALZERO;

    // capture the start time of processing a path data
    start = System.currentTimeMillis();

    // for each path
    path_ids = PIB.getAllPathIdsThatTraverseSLP(address, service_level);

    for (int index1 = INITIALZERO; index1 < path_ids.size(); index1++){

      // for each link
      myPathId = (Integer)path_ids.elementAt(index1);

      SLPs = PIB.getSLPsOfPath(myPathId.intValue());

      for (int index2 = INITIALZERO; index2 < SLPs.size(); index2++){

        mySLP = (SLP)SLPs.elementAt(index2);

        // add delay to total delay
        totalDelay = totalDelay + mySLP.getDelay();

        // add loss rate to total loss rate
        totalLossRate = totalLossRate + mySLP.getLossRate();

        // find min throughput
        throughput = mySLP.getThroughput();

        targetThroughput = mySLP.getTargetThroughput();

        throughputRemaining = targetThroughput - throughput;
        if (throughputRemaining < minThroughputRemaining ||
                                        minThroughputRemaining ==
INITIALZERO){
          minThroughputRemaining = throughputRemaining;
        }

      }


PIB.setEffectiveQoSOfPath(myPathId.intValue(),totalDelay,totalLossRate,

minThroughputRemaining);
      totalDelay = INITIALZERO;
      totalLossRate = INITIALZERO;
      minThroughputRemaining = INITIALZERO;
    }

    // capture the path data processing finish time
    finish = System.currentTimeMillis();
    gui.sendText("Server: determineEffectiveQoSForPaths: Time required
="
      +(finish-start)+" milliseconds.");
  }
  /**
   * Returns the String representation of this Server.
   * @return The String representation of this Server.
   */
```

```
  public String toString(){
    return "Server";
  }
}
```

```java
package saam.server;

import saam.net.*;

/**
 * The <em>SLP</em> class stores the QoS parameters that describe that
status
 * of a service level pipe. This class is used as a standard way to pass
this
 * information back and forth between the server and the server's path
 * information base.
 */
class SLP {
  /** The address assigned to the router interface that contains this
slp. */
  private IPv6Address address = new IPv6Address();
  /** The service level of this particular slp. */
  private int service_level = 0;
  /** The average delay experience by packets placed into this queue. */
  private int delay = 0;
  /** The average rate of packet loss experience by flows using this
queue. */
  private int loss_rate = 0;
  /** The rate of data passing through this slp. */
  private int throughput = 0;
  /** The negotiated data rate planned for this slp. */
  private int target_throughput = 0;

  /**
   * Constructs a SLP object without any arguments.
   */
  public SLP(){}

  /**
   * Constructs a SLP object using the parameters that are passed.
   * @param address The address assigned to the router interface that
contains
   * this slp.
   * @param service_level The service level of this particular slp.
   * @param delay The average delay experience by packets placed into
this
   * queue.
   * @param loss_rate The average rate of packet loss experience by
flows using
   * this queue.
   * @param throughput The rate of data passing through this slp.
   * @param target_throughput The negotiated data rate planned for this
slp.
   */
  public SLP(IPv6Address address, int service_level, int delay,
             int loss_rate, int throughput, int target_throughput){
    this.address = address;
    this.service_level = service_level;
    this.delay = delay;
    this.loss_rate = loss_rate;
    this.throughput = throughput;
    this.target_throughput = target_throughput;
  }

  /**
   * Assigns the value of what router interface this slp is assigned to.
```

```
   * @param address The address assigned to the router interface that
contains
   * this slp.
   */
  public void setAddress(IPv6Address address){
    this.address = address;
  }

  /**
   * Assigns the value of the service level of this particular slp.
   * @param service_level The service level of this particular slp.
   */
  public void setServiceLevel(int service_level){
    this.service_level = service_level;
  }

  /**
   * Assigns the value of the average delay experience by packets placed
into
   * this queue.
   * @param delay The average delay experience by packets placed into
this
   * queue.
   */
  public void setDelay(int delay){
    this.delay = delay;
  }

  /**
   * Assigns the value of the average rate of packet loss experience by
flows
   * using this queue.
   * @param loss_rate The average rate of packet loss experience by
flows using
   * this queue.
   */
  public void setLossRate(int loss_rate){
    this.loss_rate = loss_rate;
  }

  /**
   * Assigns the value of the rate of data passing through this slp.
   * @param throughput The rate of data passing through this slp.
   */
  public void setThroughput(int throughput){
    this.throughput = throughput;
  }

  /**
   * Assigns the value of the negotiated data rate planned for this slp.
   * @param target_throughput The negotiated data rate planned for this
slp.
   */
  public void setTargetThroughput(int target_throughput){
    this.target_throughput = target_throughput;
  }

  /**
   * Retrieves the value of the address assigned to the router interface
that
   * contains this slp.
```

```
     * @returns address The address assigned to the router interface that
     * contains this slp.
     */
    public IPv6Address getAddress(){
      return address;
    }

    /**
     * Retrieves the value of the service level of this particular slp.
     * @returns service_level The service level of this particular slp.
     */
    public int getServiceLevel(){
      return service_level;
    }

    /**
     * Retrieves the value of the average delay experience by packets
placed into
     * this queue.
     * @returns delay The average delay experience by packets placed into
this
     * queue.
     */
    public int getDelay(){
      return delay;
    }

    /**
     * Retrieves the value of the average rate of packet loss experience
by flows
     * using this queue.
     * @returns loss_rate The average rate of packet loss experience by
flows
     * using this queue.
     */
    public int getLossRate(){
      return loss_rate;
    }

    /**
     * Retrieves the value of the rate of data passing through this slp.
     * @returns throughput The rate of data passing through this slp.
     */
    public int getThroughput(){
      return throughput;
    }

    /**
     * Retrieves the value of the negotiated data rate planned for this
slp.
     * @returns target_throughput The negotiated data rate planned for
this slp.
     */
    public int getTargetThroughput(){
      return target_throughput;
    }

    /**
     * Generates the string representation of this service level pipe.
     * @returns slp The string representation of this service level pipe.
     */
```

```
  public String toString(){
    String slp = "\ninterface:"+address+", SLP "+service_level
      +": D = "+delay+"ms, LR = "+loss_rate+"%, T = "
      +throughput+"kbps, target T = "+target_throughput;
    return slp;
  }

}
```

```
package saam.server;

import saam.net.*;

/**
 * The <em>SLPSequence</em> class stores sequence information to
identify the
 * order of service level pipes that make up a path. This class is used
as a
 * standard way to pass this information back and forth between the
server and
 * the server's path information base.
 */
class SLPSequence {
  /** The level of service for which paths are being built for. */
  private int service_level;
  /** The node_id of the source of the SLP. */
  private int source_router;
  /** The id of physical link over which this SLP assigned. */
  private IPv6Address link_id;
  /** The number assigned to specify the sequence of this SLP in the
path. */
  private int sequence_number;
  /** The id assigned to the path. */
  private int path_id;

  /**
   * Constructs a SLPSequence object.
   */
  public SLPSequence(){
  }

  /**
   * Constructs a SLPSequence object.
   * @param service_level The level of service for which paths are being
built.
   * for.
   * @param source_router The node_id of the source of the SLP.
   * @param address The IPv6 address of an interface on the link.
   * @param sequence_number The number assigned to specify the sequence
of this
   * SLP in the path.
   * @param path_id The id assigned to the path.
   */
  public SLPSequence(int service_level, int source_router,
    IPv6Address address, int sequence_number, int path_id){
    this.service_level = service_level;
    this.source_router = source_router;
    this.link_id = address.getNetworkAddress();
    this.sequence_number = sequence_number;
    this.path_id = path_id;
  }

  /**
   * Sets the service level of the SLP.
   * @param service_level The level of service for which paths are being
built.
   * for.
   */
  public void setServiceLevel(int service_level){
    this.service_level = service_level;
```

```
  }

  /**
   * Sets the node id of the source router of this SLP.
   * @param source_router The node_id of the source of the SLP.
   */
  public void setSourceRouter(int source_router){
    this.source_router = source_router;
  }

  /**
   * Sets the link id of the SLP.
   * @param address The IPv6 address of an interface on the link.
   */
  public void setLinkId(IPv6Address link_id){
    this.link_id = link_id;
  }

  /**
   * Sets the sequence number of the SLP in the path.
   * @param sequence_number The number assigned to specify the sequence
of this
   * SLP in the path.
   */
  public void setSequenceNumber(int sequence_number){
    this.sequence_number = sequence_number;
  }

  /**
   * Sets the id of the path for which this SLP sequence object is a
part of.
   * @param path_id The id assigned to the path.
   */
  public void setPathId(int path_id){
    this.path_id = path_id;
  }

  /*
   * Retrieves the service level of this SLP.
   * @returns service_level The level of service for which paths are
being
   * built for.
   */
  public int getServiceLevel(){
    return service_level;
  }

  /*
   * Retrieves the source router of this SLP.
   * @returns source_router The node_id of the source of the SLP.
   */
  public int getSourceRouter(){
    return source_router;
  }

  /*
   * Retrieves the link id of this SLP.
   * @returns link_id The network address of the segment that this
interface
   * connects.
   */
```

```
  public IPv6Address getLinkId(){
    return link_id;
  }

  /*
   * Retrieves the sequence number of this SLP in the path.
   * @returns sequence_number The number assigned to specify the
sequence of
   * this SLP in the path.
   */
  public int getSequenceNumber(){
    return sequence_number;
  }

  /*
   * Retrieves the path id of this SLP.
   * @returns path_id The id assigned to the path.
   */
  public int getPathId(){
    return path_id;
  }

  public String toString(){
    String slps = "SL:"+service_level+", SourceRouter:"+source_router+",
link_id:"
      +link_id+", sequence#:"+sequence_number+", in path:"+path_id;
    return slps;
  }

}
```

APPENDIX J. UTIL PACKAGE SOURCE CODE

```
package saam.util;

/**
 * Array is a utility class that contains static methods for
 * manipulating arrays of various data types. This works well
 * on arrays of primitive data types; but, unfortunately,
 * it was discovered that the methods manipulating Object
 * arrays in this class eliminate the possibility of polymorphism.
 *<p>
 * This is because the only way to dynamically extend an array in
 * Java is to create a new array of different size and assign it to
 * the original array reference ("The Java Programming Language",
 * 2ed - Ken Arnold, James Gosling).  If we create a new array
 * of type Object, we lose the identity of the original array,
 * thereby disallowing a cast of the method result back to the
 * original type, which means the original type can't be of any type
 * but those types that are recognized by the methods of this class.
 *<p>
 * This implies that we must know the type of the Object array we
 * are recreating ahead of time.  Meaning this class would have
 * to contain a method for every possible Object in order to be
 * useful to the language, which is clearly impractical.<p>
 *
 * On the other hand, is losing polymorphism worth the increase
 * in performance associated with array manipulation?  We can add
 * Objects to an Object Array, and we can cast the Objects within
 * the Array when we retreive them.
 *<p>
 * The jury is still out on this one.<p>
 *
 * At any rate, <p>
 *
 *   If you're looking to dynamically expand arrays of primitive
data<br>
 *    types, this is your class.  <br>
 *<br>
 *   If you're looking to dynamically expand arrays of Objects,
while<br>
 *    maintaining the ability to polymorph the arrays themselves,<br>
 *    look to java.util.Vector.<br>
 *<br>
 *   If you wish to dynamically expand arrays of Objects, and don't<br>
 *    mind inserting elements into a generic Object array, this
class<br>
 *    will work and may be faster than Vector.<br>
 *
 */
public final class Array {

   /**
   * Returns an array of bytes that is a subarray of
   * the given byte array.  If you're familiar
   * with the substring() method in java.lang.String, you'll know
   * how to use the methods in this class.
   * @param array The array you desire a subset of.
   * @param beginIndex The beginning index, inclusive.
   * @param endIndex  The ending index, exclusive.
   * @return A new non-null byte array that is a subarray
   *         of this byte array in the range (beginIndex,
   *         endIndex-1) inclusive.
   */
```

```
  public static byte[] getSubArray(
            byte[] array, int beginIndex, int endIndex){
        byte[] subArray = new byte[endIndex-beginIndex];
        System.arraycopy(array,beginIndex,subArray,0,endIndex-
beginIndex);
        return subArray;
  }

   /**
   * Returns an array of ints that is a subarray of
   * the given int array.  If you're familiar
   * with the substring() method in java.lang.String, you'll know
   * how to use the methods in this class.
   * @param array The array you desire a subset of.
   * @param beginIndex The beginning index, inclusive.
   * @param endIndex  The ending index, exclusive.
   * @return A new non-null int array that is a subarray
   *         of this int array in the range (beginIndex,
   *         endIndex-1) inclusive.
   */
  public static int[] getSubArray(
            int[] array, int beginIndex, int endIndex){

        int[] subArray = new int[endIndex-beginIndex];
        System.arraycopy(array,beginIndex,subArray,0,endIndex-
beginIndex);
        return subArray;
  }

   /**
   * Returns an array of Objects that is a subarray of
   * the given Object array.  If you're familiar
   * with the substring() method in java.lang.String, you'll know
   * how to use the methods in this class.
   * @param array The array you desire a subset of.
   * @param beginIndex The beginning index, inclusive.
   * @param endIndex  The ending index, exclusive.
   * @return A new non-null Object array that is a subarray
   *         of this Object array in the range (beginIndex,
   *         endIndex-1) inclusive.
   */
  public static Object[] getSubArray(
            Object[] array, int beginIndex, int endIndex){

        Object[] subArray = new Object[endIndex-beginIndex];
        System.arraycopy(array,beginIndex,subArray,0,endIndex-
beginIndex);
        return subArray;
  }

   /**
   * Returns an array of bytes that is a concatenation of
   * the given byte arrays.<p>
   *     first  = index[0]..index[first.length-1],
   *     second = index[first.length]..
   *              index[first.length+second.length-1].
   * @param first The first array you wish to concatenate.
   * @param second The second array you wish to concatenate.
   * @return A new non-null byte array that is a concatenation
   *         of first and second.
   */
```

```
public static byte[] concat(byte[] first, byte[] second)
          throws IndexOutOfBoundsException {
  int length1,length2;
  try{
    length1 = first.length;
  }catch(NullPointerException npe){
    try{
      //first null, second not
      length2 = second.length;
      return second;
    }catch(NullPointerException npe2){
      //both null, now what?
      return second;//?? arbitrary since both null
    }
  }
  try{
    length2=second.length;
  }catch(NullPointerException npe){
    //first not null, but second is
    return first;
  }
  //both not null
  byte[] concatArray = new byte[length1+length2];
  System.arraycopy(first,0,concatArray,0,length1);
  System.arraycopy(second,0,concatArray,length1,length2);
  return concatArray;
}//concat(byte[] first, byte[] second)

 /**
 * Returns an array of ints that is a concatenation of
 * the given int arrays.<p>
 *     first  = index[0]..index[first.length-1],
 *     second = index[first.length]..
 *              index[first.length+second.length-1].
 * @param first The first array you wish to concatenate.
 * @param second The second array you wish to concatenate.
 * @return A new non-null int array that is a concatenation
 *         of first and second.
 */
public static int[] concat(int[] first, int[] second)
          throws IndexOutOfBoundsException {
  int length1,length2;
  try{
    length1 = first.length;
  }catch(NullPointerException npe){
    try{
      //first null, second not
      length2 = second.length;
      return second;
    }catch(NullPointerException npe2){
      //both null, now what?
      return second;//?? arbitrary since both null
    }
  }
  try{
    length2=second.length;
  }catch(NullPointerException npe){
    //first not null, but second is
    return first;
  }
  //both not null
```

```
    int[] concatArray = new int[length1+length2];
    System.arraycopy(first,0,concatArray,0,length1);
    System.arraycopy(second,0,concatArray,length1,length2);
    return concatArray;
  }//concat(int[] first, int[] second)

   /**
   * Returns an array of bytes that is a concatenation of
   * the given byte and byte array.<p>
   *     theByte = index[0],
   *     second = index[1]..
   *              index[second.length].
   * @param theByte The byte you wish to concatenate.
   * @param array The second array you wish to concatenate.
   * @return A new non-null byte array that is a concatenation
   *         of theByte and array.
   */
  public static byte[] concat(byte theByte, byte[] array)
            throws IndexOutOfBoundsException {
     try{
        byte[] newArray = new byte[array.length+1];
        newArray[0]=theByte;
        System.arraycopy(array,0,newArray,1,array.length);
        return newArray;
     }catch(NullPointerException npe){
       byte[] newArray = new byte[1];
       newArray[0] = theByte;
       return newArray;
     }
  }//concat(byte theByte, byte[] array)

   /**
   * Returns an array of ints that is a concatenation of
   * the given int and int array.<p>
   *     theInt = index[0],
   *     second = index[1]..
   *              index[second.length].
   * @param theInt The int you wish to concatenate.
   * @param array The second array you wish to concatenate.
   * @return A new non-null int array that is a concatenation
   *         of theInt and array.
   */
  public static int[] concat(int theInt, int[] array)
            throws IndexOutOfBoundsException {
     try{
        int[] newArray = new int[array.length+1];
        newArray[0]=theInt;
        System.arraycopy(array,0,newArray,1,array.length);
        return newArray;
     }catch(NullPointerException npe){
       int[] newArray = new int[1];
       newArray[0] = theInt;
       return newArray;
     }
  }//concat(int theInt, int[] array)

   /**
   * Returns an array of bytes that is a concatenation of
   * the given byte and byte array.<p>
   *     second = index[0]..
   *              index[second.length-1].
```

```
 *     theByte = index[second.length],
 * @param array The second array you wish to concatenate.
 * @param theByte The byte you wish to concatenate.
 * @return A new non-null byte array that is a concatenation
 *         of array and theByte.
 */
public static byte[] concat(byte[] array, byte theByte){
   try{
      byte[] newArray = new byte[array.length+1];
      newArray[array.length]=theByte;
      System.arraycopy(array,0,newArray,0,array.length);
      return newArray;
   }catch(NullPointerException npe){
     byte[] newArray = new byte[1];
     newArray[0] = theByte;
     return newArray;
   }
}//concat(byte[] array, byte theByte)

 /**
 * Returns an array of ints that is a concatenation of
 * the given int and int array.<p>
 *     second = index[0]..
 *              index[second.length-1].
 *     theInt = index[second.length],
 * @param array The second array you wish to concatenate.
 * @param theInt The int you wish to concatenate.
 * @return A new non-null int array that is a concatenation
 *         of array and theInt.
 */
public static int[] concat(int[] array, int theInt){
   try{
      int[] newArray = new int[array.length+1];
      newArray[array.length]=theInt;
      System.arraycopy(array,0,newArray,0,array.length);
      return newArray;
   }catch(NullPointerException npe){
     int[] newArray = new int[1];
     newArray[0] = theInt;
     return newArray;
   }
}//concat(int[] array, int theInt)
 /**
 * Returns an array of ints that is a concatenation of
 * the given long and long array.<p>
 *     second = index[0]..
 *              index[second.length-1].
 *     theInt = index[second.length],
 * @param array The second array you wish to concatenate.
 * @param theInt The long you wish to concatenate.
 * @return A new non-null long array that is a concatenation
 *         of array and theInt.
 */
public static byte[] concat(long first, long second){
   byte[] firstArray = PrimitiveConversions.getBytes(first);
   byte[] secondArray = PrimitiveConversions.getBytes(second);
   return concat(firstArray,secondArray);
}//concat(long[] array, long theInt)

/**
 * Returns an array of Objects that is a concatenation of
```

```java
 * the given Object and Object array.<p>
 *     second = index[0]..
 *              index[second.length-1].
 *     theObject = index[second.length],
 * @param array The second array you wish to concatenate.
 * @param theObject The Object you wish to concatenate.
 * @return A new non-null Object array that is a concatenation
 *         of array and theObject.
 */
public static Object[] concat(Object[] array, Object theObject){
   try{
      Object[] newArray = new Object[array.length+1];
      newArray[array.length]=theObject;
      System.arraycopy(array,0,newArray,0,array.length);
      return newArray;
   }catch(NullPointerException npe){
     Object[] newArray = new Object[1];
     newArray[0] = theObject;
     return newArray;
   }
}//concat(Object[] array, Object theObject)

 /**
 * Returns an array of bytes with theByte inserted into
 * array at position.<p>  This method automatically expands
 * array if position > the length of array.
 * @param array The byte you wish to add to the array.
 * @param theByte The second array you wish to concatenate.
 * @param position The position at which the byte will be added.
 * @return A new non-null byte array with theByte inserted into
 * array at position.<p>
 */
public static byte[] replaceElementAt(byte theByte, byte[] array,
                                int position)
          throws IndexOutOfBoundsException {

      int length = 0;
      try{
        length = array.length;
        //if the exception is thrown, we know length = 0,
        //so we do nothing with the exception
      }catch(NullPointerException npe){}

      if(position<0){
        throw new IndexOutOfBoundsException(
          "Can't add "+theByte+" to position "+position+".");
      }else if(position<length){
        array[position]=theByte;
        return array;
      }else{
        //automatically expand the array if the insertion
        //point (position) is greater than the length of
        //the old array.
        int newlength=position+1;
        byte[] concatArray = new byte[newlength];
        concatArray[position]=theByte;
        System.arraycopy(array,0,concatArray,0,length);
        return concatArray;
      }//else
}//replaceElementAt()
```

```
    /**
     * Returns an array of ints with theInt inserted into
     * array at position.<p>  This method automatically expands
     * array if position > the length of array.
     * @param array The int you wish to add to the array.
     * @param theInt The second array you wish to concatenate.
     * @param position The position at which the int will be added.
     * @return A new non-null int array with theInt inserted into
     * array at position.<p>
     */
   public static int[] replaceElementAt(int theInt, int[] array,
                                   int position)
            throws IndexOutOfBoundsException {

        int length = 0;
        try{
          length = array.length;
          //if the exception is thrown, we know length = 0,
          //so we do nothing with the exception
        }catch(NullPointerException npe){}

        if(position<0){
          throw new IndexOutOfBoundsException(
            "Can't add "+theInt+" to position "+position+".");
        }else if(position<length){
          array[position]=theInt;
          return array;
        }else{
          //automatically expand the array if the insertion
          //point (position) is greater than the length of
          //the old array.
          int newlength=position+1;
          int[] concatArray = new int[newlength];
          concatArray[position]=theInt;
          System.arraycopy(array,0,concatArray,0,length);
          return concatArray;
        }//else
   }//replaceElementAt()

    /**
     * Returns an array of Objects with theObject inserted into
     * array at position.<p>  This method automatically expands
     * array if (position > the length of array).
     * @param theObject The Object you wish to add to the array.
     * @param array The array to be added to.
     * @param position The position at which the Object will be added.
     * @return A new non-null Object array with theObject inserted into
     * array at position.<p>
     */
   public static Object[] replaceElementAt(Object theObject, Object[]
array,
                                   int position)
            throws IndexOutOfBoundsException {

        int length = 0;
        try{
          length = array.length;
          //if the exception is thrown, we know length = 0,
          //so we do nothing with the exception
        }catch(NullPointerException npe){}
```

```
        if(position<0){
          throw new IndexOutOfBoundsException(
            "Can't add "+theObject+" to position "+position+".");
        }else if(position<length){
          array[position]=theObject;
          return array;
        }else{
          if (length==0){array = new Object[1];}
          //automatically expand the array if the insertion
          //point (position) is greater than the length of
          //the old array.
          int newlength=position+1;
          Object[] concatArray = new Object[newlength];
          concatArray[position]=theObject;
          System.arraycopy(array,0,concatArray,0,length);
          return concatArray;
        }//else
  }//replaceElementAt()

   /**
    * Returns an array of Objects that is a concatenation of
    * the given Object arrays.<p>
    *      first  = index[0]..index[first.length-1],
    *      second = index[first.length]..
    *               index[first.length+second.length-1].
    * @param first The first array you wish to concatenate.
    * @param second The second array you wish to concatenate.
    * @return A new non-null array that is a concatenation
    *         of first and second.
    */
  public static Object[] concat(Object[] first, Object[] second)
            throws IndexOutOfBoundsException {
    int length1,length2;
    try{
      length1 = first.length;
    }catch(NullPointerException npe){
      try{
        //first null, second not
        length2 = second.length;
        return second;
      }catch(NullPointerException npe2){
        //both null, now what?
        return second;//?? arbitrary since both null
      }
    }
    try{
      length2=second.length;
    }catch(NullPointerException npe){
      //first not null, but second is
      return first;
    }
    //both not null
    Object[] concatArray = new Object[length1+length2];
    System.arraycopy(first,0,concatArray,0,length1);
    System.arraycopy(second,0,concatArray,length1,length2);
    return concatArray;
  }//concat(Object[] first, Object[] second)
}
```

```java
package saam.util;

/**
 * A utility class with static methods that performs conversions
 * from a byte array to a primitive data type and from a primitive
 * data type to a byte array.
 */
public final class PrimitiveConversions{

  /**
   * Converts a long to a byte array containing 8 bytes.
   * @param in The long to be converted.
   * @return The byte array that represents the long supplied.
   */
  public static byte[] getBytes(long in){
    int length = 8;
    byte[] result = new byte[length];
    for(int x=length-1;x>=0;x--){
      result[x] = (byte)(in>>((length-1-x)*8));
    }
    return result;
  }

  /**
   * Converts an int to a byte array containing 4 bytes.
   * @param in The int to be converted.
   * @return The byte array that represents the int supplied.
   */
  public static byte[] getBytes(int in){
    int length = 4;
    byte[] result = new byte[length];
    for(int x=length-1;x>=0;x--)
      result[x] = (byte)(in>>((length-1-x)*8));
    return result;
  }

  /**
   * Converts an int to a byte array containing the number of bytes
   * requested.
   * @param in The int to be converted.
   * @return The byte array that represents as many bytes of
   * the int as are requested.  This method is intended to allow
   * smaller ints to be stuffed into a fewer number of bytes; but,
   * it is up to the user of this method to determine whether or not
   * the int can be accurately represented by the number of bytes
   * requested.
   */
  public static byte[] getBytes(int in, int howManyBytes){
    int length = 4;
    if(howManyBytes>0&&howManyBytes<5){
      length = howManyBytes;
    }
    byte[] result = new byte[length];
    for(int x=length-1;x>=0;x--)
      result[x] = (byte)(in>>((length-1-x)*8));
    return result;
  }

  /**
   * Converts a short to a byte array containing two bytes.
   * @param in The short to be converted.
```

```
 * @return The byte array that represents the short supplied.
 */
public static byte[] getBytes(short in){
  int length = 2;
  byte[] result = new byte[length];
  for(int x=length-1;x>=0;x--)
    result[x] = (byte)(in>>((length-1-x)*8));
  return result;
}

/**
 * Converts a byte array of 1 or 2 bytes into a short.  If the
 * array is null, this method returns zero.
 * @param array The array to be converted.
 * @return The result of converting the array to a short.
 */
public static short getShort(byte[] array){
  int length = 0;
  try{
    length = array.length;
  }catch(NullPointerException npe){
    return 0;
  }
  if (length>2)length=2;
  short s = 0;
  for(int x=length-1;x>=0;x--){
    s|=((array[x]&0xFF)<<((length-1-x)*8));
  }
  return s;
}

/**
 * Converts a byte array of 1 to 4 bytes into an int.  If the
 * array is null, this method returns zero.
 * @param array The array to be converted.
 * @return The result of converting the array to a int.
 */
public static int getInt(byte[] array){
  int length = 0;
  try{
    length = array.length;
  }catch(NullPointerException npe){
    return 0;
  }
  if (length>4)length=4;
  int s = 0;
  for(int x=length-1;x>=0;x--){
    s|=((array[x]&0xFF)<<((length-1-x)*8));
  }
  return s;
}

/**
 * Converts a byte array of 1 to 8 bytes into an long.  If the
 * array is null, this method returns zero.
 * @param array The array to be converted.
 * @return The result of converting the array to a long.
 */
public static long getLong(byte[] array){
  int length = 0;
  try{
```

```
      length = array.length;
    }catch(NullPointerException npe){
      return 0;
    }
    if (length>8)length=8;
    long s = 0;
    for(int x=length-1;x>=0;x--){
      s|=((long)(array[x]&0xFF)<<((length-1-x)*8));
    }
    return s;
  }
}
```

```
// Dean Vrable - CS3973 - March, 1998

package saam.util;

/**
 * A thread-safe circular linked list implementation of a queue.
Adapted
 * from Sethi's <em>Programming Languages</em> book
 * http://store.awl.com/scatalog/offer.mhtml?sho=0-201-59065-4&n=1
 */

public class Queue {

      private Object obj;
      private Queue next;
      private int count = 0;

      /**
       * A static method that returns an empty <code>Queue</code>
       * @return An empty <code>Queue</code>.
       */
      public static Queue emptyQ() {

            Queue q = new Queue();
            q.obj = null;
            q.next = q;
            return q;
      }

      /**
       * Inserts an <code>Object</code> at the tail of the
       * <code>Queue</code>.
       * @param o The <code>Object</code> to be inserted into the
       * <code>Queue</code>.
       * @return A new <code>Queue</code> with this <code>Object</code>
       * inserted at the tail.
       */
      public synchronized Queue enqueue(Object o) {

            Queue q = new Queue();

            q.obj = null;
            q.next = this.next;

            this.obj = o;
            this.next = q;
            return q;
      }

      /**
       * Peeks at the next <code>Object</code> to be removed from the
       * <code>Queue</code>.
       * @return The <code>Object</code> at the head of the
       * <code>Queue</code>.
       */
      public synchronized Object peek() {

//    return this.next.obj;
    return this.obj;
      }
```

```
    /**
     * Removes the <code>Object</code> at the head of the
     * <code>Queue</code>.
     * @return The <code>Queue</code> with the Object at the
 *         head removed.
     */
    public synchronized Queue dequeue() {

          Queue front = this.next;
          this.next = front.next;
          return this;
    }

    /**
     * Determines if the <code>Queue</code> is empty.
     * @return     true if the queue is empty
     */
    public synchronized boolean isemptyQ() {

          return this == this.next;
    }

    /**
     * Returns the number of objects in this <code>Queue</code>.
     * @return The number of objects in this <code>Queue</code>.
     */
    public synchronized int getCount(){
          count=0;
//go around the circle until you arrive back at "this"
          for (Queue q = this.next; q != this; q = q.next){
                count++;
          }
          return count;
    }

    /**
     * A <code>String</code> representation of all objects in this
     * <code>Queue</code>.
     * @return A String representing of all objects in this
     * <code>Queue</code>
     */
    public synchronized String toString() {

          String s = new String(getCount()+" elements [");

          for (Queue q = this.next; q != this; q = q.next){
                s += q.obj.toString();
          }
          return s + "]";
    }
}
```

```java
package saam.util;

import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/**
 * A simple Graphical User Interface that consists of a JPanel with
 * a JTextArea and a JTextField that can be updated with the sendText
 * and setTextField methods respectively.
 */
public class SAAMRouterGui extends JPanel {

   public static Hashtable instances = new Hashtable();
   public static Vector titles = new Vector();
   private boolean show = true;
   private int rows = 6;
   private int cols = 25;
   private JTextArea messages = new
     JTextArea(rows,cols);
   private JTextField tf = new JTextField();
   private String title;

   public SAAMRouterGui(String title){
      super(new BorderLayout());
      this.title=title;
      titles.add(title);
      instances.put(title,this);
      add(messages, BorderLayout.CENTER);
      add(tf,BorderLayout.SOUTH);
      messages.setBackground(new Color(150,172,190));
      tf.setBackground(new Color(150,150,255));
      tf.setFont(new Font("serif",Font.BOLD,12));
      setLocation(0,0);
      if(show)setVisible(true);
   }
   public static SAAMRouterGui getInstance(String name){
     return (SAAMRouterGui)instances.get(name);
   }
   public static Vector getTitles(){
     return titles;
   }
   public void sendText(String message){
      if(show)messages.append(message + "\n");
   }
   public void setTextField(String message){
      if(show)tf.setText(message);
   }
   public JTextArea getTextArea(){
      return messages;
   }
   public JTextField getTextField(){
      return tf;
   }
   public void setTextFieldFontSize(int size){
      if(show)tf.setFont(new Font("serif",Font.BOLD,size));
   }
   public String toString(){
     return title;
   }
}
```

REFERENCES

[1] Steinke, S., "ATM and Alternatives in the Wide Area Backbone." *Network Magazine*, pp. 36-42, July 1999.

[2] Karvé, Anita, "Lesson 119: IP Quality of Service," *Network Magazine*, pp. 36-42, June 1998.

[3] Rumbaugh, James, Jacobson, Ivar, and Booch, Grady, *The Unified Modeling Language Reference Manual*, Addison Wesley Longman, Inc., 1999.

[4] Sackett, George C. and Metz, Christopher Y., *ATM and Multiprotocol Networking*, McGraw-Hill, 1997.

[5] Peterson, Larry L. and Davie, Bruce S., *Computer Networks, A Systems Approach.* Morgan Kaufmann, 1997.

[6] Floyd, Sally and Jacobson, Van, "Link-sharing and resource management models for packet networks," *IEEE/ACM Transactions on Networking*, 3(4):365-386, August 1995.

[7] Bennett, Jon C.R. and Zhang, Hui, "Hierarchial Parcket Fair Queueing Algorithms," *IEEE/ACM Transactions on Networking*, 5(5):675-689, October 1997.

[8] Xie, Geoffrey G. and Lam, Simon S., "Real-Time Block Transfer Under a Link Sharing Hierarchy," *IEEE/ACM Transactions on Networking*, 6(1):30-41, February 1998.

[9] Xie, Geoffrey G. and Lam, Simon S., "Delay Guarantee of Virtual Clock Server," *IEEE/ACM Transactions on Networking*, 3(6):683-689, December 1995.

[10] Lam, Simon S. and Xie, Geoffrey G., "Group Priority Scheduling," *IEEE/ACM Transactions on Networking*, 5(2):205-218, April 1997.

[11] Xie, Geoffrey G., Hensgen, Debra, Kidd, Taylor, and Yarger, John, "SAAM: An Integrated Network Architecture for Integrated Services," paper presented at the 6th IEEE/IFIP International Workshop on Quality of Service, Napa, CA, [http://www.cs.nps.navy.mil/people/faculty/xie/pub]. May 1998.

[12] Xie, Geoffrey G., Hensgen, Debra, Kidd, Taylor, and Yarger, John, "Efficient Management of Integrated Services Using a Path Information Base." [http://www.cs.nps.navy.mil/people/faculty/xie/pub]. 14 May 1998.

[13] Larman, Craig, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Prentice Hall, Inc., 1998.

[14] Berg, Daniel J. and Fritzinger, Steven, *Advanced Techniques for Java Developers, Proven Solutions from Leading Java Experts*, John Wiley & Sons, Inc., 1997.

[15] Hunter, Jason and Crawford, William, *Java Servlet Programming*, O'Reilly & Associates, Inc., 1998.

[16] Sridharan, Prashant, *Advanced Java Networking*, Prentice Hall, Inc., 1997.

[17] Hughes, Merlin, et al, *Java Network Programming*, Manning Publications Co., 1999.

[18] Boone, Barry, *Java Certification Exam Guide For Programmers and Developers*, McGraw-Hill Companies, Inc., 1997.

[19] Roberts, Simon and Heller, Philip, *Java 1.1 Certification Study Guide*, SYBEX Inc., 1997.

[20] Murphy, David M., *Building an Active Node on the Internet*, Master's Thesis, Massachusetts Institute of Technology, Boston, Massachusetts, May 1997.

[21] Lange, Danny B. and Oshima, Mitsuru, *Programming and Deploying Java Mobile Agents with Aglets,* Addison Wesley Longman, Inc., 1998.

[22] Heller, Philip and Roberts, Simon, Java 1.2 Developer's Handbook, SYBEX Inc., 1999.

[23] Calvert, Kenneth L., et al, "Directions in Active Networks," *IEEE Communications*, 36(10), Oct. 1998.

[24] Krupczak, Bobby, et al, "Implementing Communication Protocols in Java", *IEEE Communications*, 36(10), Oct. 1998.

[25] Ricciulli, Livio, "ANETD: Active NETworks Daemon (v1.0)." [http://www.csl.sri.com/ancors/anetd/docs]. Aug 1998.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center..2
   8725 John J. Kingman Road, Ste 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. Director, Training and Education 1
   MCCDC, Code C46
   1019 Elliot Road
   Quantico, VA 22134-5027

4. Director, Marine Corps Research Center 2
   MCCDC, Code C40RC
   2040 Broadway Street
   Quantico, VA 22134-5107

5. Director, Studies and Analysis Division 1
   MCCDC, Code C45
   3300 Russell Road
   Quantico, VA 22134-5130

6. Marine Corps Representative 1
   Naval Postgraduate School
   Bldg. 330, IN-116
   555 Dyer Road
   Monterey, CA 93943

7. Marine Corps Tactical Systems Support Activity......................................1
   Technical Advisory Branch
   Attn: Maj J. C. Cummiskey
   Box 555171
   Camp Pendleton, CA 92055-5080

8. Chairman,Code CS...1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93940-5000

9. Dr. Geoffrey Xie..1
Computer Science Department, Code CS
Naval Postgraduate School
Monterey, California 93943-5100

10. Dr. Debra Hensgen...1
Computer Science Department, Code CS
Naval Postgraduate School
Monterey, California 93943-5100

11. Mr. Cary Colwell..1
Computer Science Department, Code CS
Naval Postgraduate School
Monterey, California 93943-5100

12. Dr. Donald Brutzman...1
Undersea Research Department, Code UW
Naval Postgraduate School
Monterey, California 93943-5100

13. Dr. Gordon Bradley..1
Operational Research Department, Code OR
Naval Postgraduate School
Monterey, California 93943-5100

14. Dr. Harry K. Edwards..1
Computer Science Department
University of Michigan
Flint, Michigan 48504

15. Captain Dean J. Vrable..3
6225 Chesaning Rd.
Chesaning, MI 48616

16. Captain John W. Yarger...3
P.O. Box 114
Brisbin, PA 16620